Etude et conception d'un réseau sur puce dynamiquement adaptable pour la vision embarquée

Nicolas Ngan

UNIVERSITÉ PARIS-EST

ESIEE PARIS

SAFRAN Sagem

UNIVERSITE PARIS-EST : Ecole doctorale MSTIC

THÈSE

pour obtenir le grade de Docteur de l'Université Paris-Est
Spécialité Informatique

Etude et conception d'un réseau sur puce dynamiquement adaptable pour la vision embarquée

Nicolas Ngan

présentée et soutenue publiquement le 09 décembre 2011

Composition du Jury:

Lionel TORRES	Professeur, LIRMM, Université Montpellier 2	Rapporteur
Fan YANG	Professeur, Le2i, Université Bourgogne	Rapporteur
Mohamed AKIL	Professeur, A3SI, ESIEE	Directeur
Eva DOKLADALOVA	Professeur associé, A3SI, ESIEE	Co-directeur
Virginie FRESSE	Maître de conférence, Université St Etienne	Examinateur
Marc BOUSQUET	Directeur Optronique & TI, Sagem	Examinateur

Table des matières

Table des figures

Liste des tableaux

Glossaire

BNL	**B**as **N**iveau de **L**umière
CMOS	**C**omplementary **M**etal **O**xide **S**emiconductor
DFR	**D**ata **F**low **R**outer
FBS	**F**rame **B**uffer **S**ystem
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FWD	**F**or**W**ar**D** Mode
HDR	**H**igh **D**ynamic **R**ange
IR	**I**nfra**R**ouge
MAC	**M**ultiplieur-**AC**cumulateur
MDFR	**M**ulti **D**ata **F**low **R**ing
NOC	**N**etwork-**O**n-**C**hip
NUC	**N**on-**U**niformity **C**orrection
PE	**P**rocessing **E**lement
SAF	**S**tore **A**nd **F**orward
SGM	**S**tream **G**ate **M**anager
SOC	**S**ystem **O**n a **C**hip
SOPC	**S**ystem **O**n a **P**rogrammable **C**hip
SS	**S**tream **S**witch
SSP	**S**ingle **S**tream **P**rocessing Mode
SSF	**S**ingle **S**tream & **F**orward Mode
SWS	**S**plit **W**ormhole **S**witching
MS	**M**ulti **S**tream Mode
VCT	**V**irtual **C**ut **T**hrough
VLIW	**V**ery **L**ong **I**nstruction **W**ord
WS	**W**ormhole **S**witching

Résumé

Un équipement portable moderne intègre plusieurs capteurs d'image qui peuvent être de différents types. On peut citer en guise d'exemple un capteur couleur, un capteur infrarouge ou un capteur basse lumière. Cet équipement doit alors supporter différentes sources qui peuvent être hétérogènes en terme de résolution, de granularité de pixels et de fréquence d'émission des images. Cette tendance à multiplier les capteurs, est motivée par des besoins applicatifs dans un but de complémentarité en sensibilité (fusion des images), en position (panoramique) ou en champ de vision.

Le système doit par conséquent être capable de supporter des applications de plus en plus complexes et variées, nécessitant d'utiliser une seule ou plusieurs sources d'image.

Du fait de cette variété de fonctionnalités embarquées, le système électronique doit pouvoir s'adapter constamment pour garantir des performances en terme de latence et de temps de traitement en fonction des applications, tout en respectant des contraintes d'encombrement.

Nous proposons dans cette thèse un nouveau *réseau de communication sur puce* (NoC) pour un système sur puce (SoC) dédié à la vision. Ce réseau permet de gérer dynamiquement différents types de flux en parallèle en auto-adaptant le chemin de donnée entre les unités de calcul, afin d'éxécuter de manière efficace différentes applications.

La proposition d'une nouvelle structure de paquets de données, facilite les mécanismes d'adaptation du système grâce à la combinaison d'instructions et de données à traiter dans un même paquet.

Nous proposons également un système de mémorisation de trames à adressage indirecte, capable de gérer dynamiquement plusieurs trames image de différentes sources d'image. Cet adressage indirect est réalisé par l'intermédiaire d'une couche d'abstraction matérielle qui se charge de traduire des requêtes de lecture et d'écriture, réalisées suivant des indicateurs de la trame requise (source de l'image, indice temporel et dernière opération effectuée).

Afin de valider notre proposition, nous définissons une nouvelle architecture, appelée *Multi Data Flow Ring* (MDFR) basée sur notre réseau avec une topologie en anneau. Les performances de cette architecture, en temps et en surface, ont été évaluées dans le cadre d'une implémentation sur une cible FPGA.

Abstract

Modern portable vision systems include several types of image sensors such as colour, low-light or infrared sensor. Such system has to support heterogeneous image sources with different spatial resolutions, pixel granularities and working frequencies. This trend to multiply sensors is motivated by needs to complete sensor sensibilities with image fusion processing techniques, or sensor positions in the system. Moreover, portable vision systems implement image applications which require several images sources with a growing computing complexity.

To face those challenges in integrating such a variety of functionnalities, the embedded electronic computing system has to adapt permanently to preserve application timing performance in latency and processing, and to respect area and low-power constraints.

In this thesis, we propose a new Network-On-Chip (NoC) adapted for a System-On-Chip (SoC) dedicated to image applications. This NoC can manage several pixel streams in parallel by adapting dynamically the datapatah between processing elements and memories. The new header packet structure enables adaptation mechanisms in routers by combining instructions and data in a same packet.

To manage efficiently the frames storage required for an application, we propose a frame buffer system with an indirect frame adressing, which is able to manage several frames from different sensors. It features a hardware abstraction layer which is in charge to collect reading and writing requests, according to specific frame indicators such as the image source ID.

The NoC has been validated in a complete processing architecture called *Multi Data Flow Ring* (MDFR) with a ring topology. The MDFR performances in time and area has been demonstrated for an FPGA target.

Remerciements

Je tiens à remercier, en premier lieu, mon directeur de thèse, Mohamed Akil. Je le remercie pour son encadrement durant ma thèse et j'ai une profonde gratitude pour tout ce qu'il m'a apporté humainement, techniquement et professionnellement. Je le remercie tout particulièrement de m'avoir introduit dans le domaine passionant des architectures de calcul pour la vision embarquée, depuis mon cursus en école d'ingénieur, et de de m'avoir permis de l'approfondir durant cette thèse.

Je remercie très chaleureusement ma co-directrice de thèse, Eva Dokladalova, qui m'a beaucoup appris pendant cette thèse. Je la remercie particulièrement pour sa disponibilité, sa rigueur scientifique et son soutien permanent tout au long de mon travail.

Je remercie très sincèrement Lionel Torres et Fan Yang de m'avoir fait l'honneur d'être les rapporteurs de mon manuscrit de thèse. Je remercie également Virginie Fresse d'avoir accepté d'être examinatrice de ma thèse. Leurs contributions respectives dans le domaine des architectures de calcul et des systèmes dédiés à la vision embarquée m'ont en effet beaucoup inspiré dans mes travaux.

Je remercie tous les membres des équipes de R&D (FPGA, traitement d'image, logiciel, optique, mécanique, etc.) de la société Sagem avec qui j'ai eu le plaisir de travailler pendant ma thèse à la fois sur les sites de Massy et d'Argenteuil. Je remercie tout particulièrement Sylvain Faure pour m'avoir fait confiance dans mon projet et François Contou-Carrère pour m'avoir encadré pendant cette thèse. Je voudrais aussi remercier Sébastien Guérin et Benoît Marcon qui m'ont soutenu pour la mise en oeuvre de ce projet. Je remercie aussi Marc Bousquet, Joël Budin, Laurent Huré et Jérôme Beauté pour leur soutien et leur confiance dans mes travaux.

Je remercie Gilles Bertrand pour m'avoir accueilli au sein du laboratoire A3SI ainsi que tous les membres (professeurs, doctorants et même stagiaire de recherche) avec qui j'ai eu le plaisir de discuter de mes travaux. Je remercie tout spécialement Thierry Grandpierre, Laurent Perroton, Harold Phelippeau, Ramzi Mahmoudi, Rostom Kachouri, Oussama Féki, Jan Bartovsky, pour leurs conseils et leur bonne humeur. J'ai également une profonde gratitude pour Geoffroy Marpeaux qui a grandement contribué à la réussite de ce projet et avec qui j'ai eu la joie de travailler.

Je remercie ma famille, mes parents, ma soeur et mon frère pour l'affection et leur soutien permanent tout au long de ma thèse.

Chapitre 1

Introduction

Les travaux de recherche menés et présentés dans ce manuscrit ont été effectués dans le cadre d'une convention CIFRE entre le *Centre d'Excellence Caméra Thermique et Portable* (CE-CTP) de la société Sagem (Groupe SAFRAN) et le *laboratoire Informatique Gaspard Monge, Equipe ESIEE A3SI*, Unité Mixte CNRS-UMLV-ESIEE (UMR 8049) de l'Université Paris-Est.

Les systèmes de vision embarqués et portables ont connu ces dernières années des progrès technologiques dans différents domaines : qualité des objectifs, performance des capteurs d'image, des afficheurs portables et de l'électronique embarquée bénéficiant des avancées en technologie d'intégration des circuits intégrés réalisés avec une finesse de gravure croissante. Cette forte dynamique des systèmes embarqués pour la vision, est stimulée par un marché qui s'est considérablement élargi dans des domaines spécialisés comme la vidéo-surveillance, et grand public comme la téléphonie mobile. Dans le secteur industriel, ces systèmes sont également impliqués pour tout système pouvant nécessiter une assistance dans le guidage à la fois dans le secteur terrestre (jumelle d'observation portable, automobile), naval et aérien (avion, hélicoptère, drone).

Un équipement portable moderne intègre plusieurs capteurs d'image qui peuvent être de différents types. On peut citer en guise d'exemple un capteur couleur [1], un capteur infrarouge [2, 3] ou un capteur basse lumière [4–6]. Cet équipement doit alors supporter différentes sources qui peuvent être hétérogènes en terme de résolution, de granularité de pixels et de fréquence d'émission des images [7]. Ces capteurs ne cessent de croître en

terme de résolution spatiale avec des tailles d'acquisition d'image de l'ordre de plusieurs dizaines de Mbits avec des débits atteignant aisément le Gbit par seconde.

Cette tendance à multiplier les capteurs, est motivée par des besoins applicatifs dans un but de complémentarité en sensibilité (fusion des images), en position (panoramique) ou en champ de vision. Le système doit par conséquent être capable de supporter des applications de plus en plus complexes et variées [8], nécessitant d'utiliser une seule ou plusieurs sources d'image. Ces applications subissent des évolutions permanentes motivées par un souci de robustesse et une volonté d'accroître le nombre de fonctionnalités dans un système pour le rendre le plus polyvalent possible.

Dans le domaine de la vision, ces applications couvrent des besoins principaux et incontournables comme la restauration d'image, l'amélioration de l'image, l'analyse d'image et la visualisation. Ces applications imposent par exemple un traitement de trames en parallèle, dans le but de les fusionner dans le domaine spatiale et temporel. En pratique, le choix des fonctionnalités d'un système moderne, est réalisé dynamiquement, suivant un besoin utilisateur et en fonction de l'environnnement.

Du fait de cette variété de fonctionnalités embarquées, le système électronique doit pouvoir s'adapter constamment pour garantir des performances en terme de latence et de temps de traitement en fonction des applications, tout en respectant des contraintes d'encombrement.

Dans ce but, nous distinguons deux niveaux d'adaptation pour ce système : le niveau *fonctionnel* et le niveau *architectural*. Le niveau fonctionnel correspond à la modification de l'ordonnancement des unités de calcul selon l'application. Le niveau architectural correspond à la modification de la structure architecturale du système électronique pour effectuer le calcul.

Pour l'étude de l'adaptation au niveau architectural, nous nous positionnons au niveau du *système sur puce* (SoC).

Même si depuis de nombreuses années, un grand nombre de solutions architecturales ont été proposées pour améliorer l'adaptabilité des unités de calcul [9, 10], un problème majeur persiste au niveau du réseau d'interconnexion qui n'est pas suffisamment adaptable, en particulier pour le transfert des flux de pixels et l'accès aux données.

De multiples solutions d'interconnexion ont été proposées [11] et utilisées de manière traditionnelle dans les système de vision comme des systèmes point-à-point ou multi-bus hiérarchique. Ces types d'interconnexion deviennent rapidement complexes à gérer avec des latences de communication croissantes. Ces latences sont dûes au fait que les systèmes sont en constante évolution en nombre d'unités de calcul et en nombre de flux de pixels, de différents types, à traiter.

Depuis une dizaine d'années [12–14], les premiers concepts et prototypes de réseau de communication sur puce, appelé *Network-on-Chip*, sont apparus. Le réseau de communication constitue une réponse actuelle pour faire face à la croissance du nombre de coeurs de traitement sur une puce.

De nombreuses études ont été effectuées depuis, pour porter et évaluer différentes solutions d'interconnexion de multi-processeurs vers cette nouvelle approche de réseau de communication. C'est le cas par exemple pour l'architecture *Chameleon* dont les communications ont été évaluées dans un contexte de réseau de processeurs sur puce [15].

La multiplicité de capteurs [16] apporte une difficulté supplémentaire à la conception du système d'interconnexion. La majorité des systèmes multi-capteurs, ciblant généralement des applications de stéréovision, de fusion ou de réhaussement HDR, utilisent des solutions de communication dédiées et ainsi peu flexibles à des modifications d'applications. Nous pouvons citer en exemple le système *StereoCam* [17] de *Philips* pour une application de stéréovision, utilisant une communication point-à-point, ou l'architecture *CRISP-DS* [18] pour une application *High Dynamic Range* (HDR), utilisant des unités de calcul dédiées dans le pipeline. Ces solutions décrivent des pipelines d'unités de calcul définies et imposées pouvant être programmables ou reconfigurables.

Le principal goulot d'étranglement réside principalement dans la gestion de multiples flux de données. Ce verrou technologique est d'autant plus critique dans le domaine de la vision embarquée multi-capteurs. En particulier, le problème de l'adaptation du chemin de données dans le contexte de multiples flux hétérogènes en parallèle n'a pas été suffisamment étudié.

La problématique repose sur la manière d'acheminer et de traiter plusieurs flux image différents en parallèle tout en garantissant les performances. Elle s'inscrit dans notre

contexte où le chemin de données doit pouvoir s'adapter de manière dynamique en fonction de l'application et des unités de calcul disponibles dans le SoC.

1.1 Contribution

Nous proposons dans cette thèse un nouveau *réseau de communication sur puce* (NoC) pour un SoC dédié à la vision multi-capteurs [19, 20]. Ce réseau permet de gérer dynamiquement différents types de flux en parallèle en auto-adaptant le chemin de donnée, afin d'éxécuter de manière efficace différentes applications.

Il est construit à partir de deux types de routeurs, *maîtres* et *esclaves*. Les paquets de données transitent principalement entre deux routeurs maîtres qui sont séparés par un réseau linéaire de routeur esclaves comportant des unités de calculs. Afin de traiter les données sur ces unités de calcul, le réseau auto-adapte le chemin de données des routeurs esclaves en fonction d'un ensemble d'opérations directement spécifié dans l'en-tête du paquet.

La proposition d'une nouvelle structure de paquets de données, facilite les mécanismes d'adaptation du système grâce aux informations contenues dans l'en-tête. Nous définissons ainsi un en-tête, contenant à la fois des informations caractéristiques de la trame image (source de l'image, indice temporel, dernière opération appliquée), des informations d'adressage (noeud source et de destination) et des commandes opératives sur la donnée. Ces commandes opératives sont constituées d'un ensemble d'instructions destiné aux routeurs pour adapter le chemin de donnée du paquet. Ces instructions décrivent une application suivant différents modes d'exécution des opérations à appliquer sur la donnée, de manière séquentielle et parallèle. Chaque instruction contient, en particulier, le type d'opération, le nombre d'itération requis, et le mode d'exécution à appliquer dans le routeur.

Ce réseau est utilisé avec la mise en oeuvre d'une nouvelle organisation mémoire dédiée et dotée de mécanismes d'adaptation à la fois en accès et en chemin de donnée, tirant pleinement profit de la nouvelle structure de paquets de données. Il s'agit plus particulièrement d'un système de mémorisation de trames à adressage indirecte, capable de gérer dynamiquement plusieurs trames image de différentes sources d'image. Cet adressage indirect est réalisé à travers une couche d'abstraction matérielle qui se charge

de traduire des requêtes de lecture et d'écriture, réalisées suivant des indicateurs de la trame requise (source de l'image, indice temporel et dernière opération effectuée). Nous montrons que cette méthode d'adressage, dans notre réseau, permet l'optimisation du stockage en mémoire, avec différents modes d'accès, et une meilleure réutilisation du système mémoire.

Afin de valider notre proposition, nous définissons une nouvelle architecture, appelée *Multi Data Flow Ring* (MDFR) basée sur notre réseau avec une topologie en anneau. Les performances de cette architecture, en temps et en surface, ont été évaluées dans le cadre d'une implémentation sur une cible FPGA avec des applications typiques d'un équipement de vision embarqué portable de la société Sagem.

1.2 Plan du manuscrit

Ce manuscrit, organisé en six chapitres, présente l'étude et la conception d'un nouveau réseau de communication sur puce, dynamiquement adaptable. Il est destiné à des équipements de vision multi-capteurs, embarqués et portables, nécessitant de supporter des applications variées en traitement d'image.

1.2.1 Système de vision embarqué portable et multi-capteurs

Le premier chapitre de ce manuscrit est dédié à la présentation des défis dans la conception d'architectures de traitement d'image pour un système moderne en vision embarqué et portable. Ce chapitre étudie dans un premiers temps, les besoins applicatifs d'un équipement de vision multi-capteurs portable et les contraintes d'un tel système. Nous présentons ensuite différentes solutions architecturales dans le domaine de la vision embarquée portable et nous analyserons les forces et faiblesses de chaque solution appliquée à notre contexte. De cette étude, nous dresserons un bilan des besoins pour la conception d'un système de vision embarquée portable et multi-capteurs.

1.2.2 Systèmes d'interconnexion dans un SoC

Le second chapitre de ce manuscrit présente une description succinte des systèmes d'interconnexions dans un système sur puce (SoC, *System-On-Chip*). Ce chapitre montre

la variété des méthodes de communication réalisables entre des unités de calcul sur une puce et souligne les limitations des solutions les plus classiques pour un système de vision. Après un rappel de la terminologie utilisée pour décrire les réseaux sur puce (NoC, *Network-On-Chip*), nous effectuons un bilan de l'utilisation de ce système d'interconnexion dans la littérature et discutons de son application pour un système de vision embarqué multi-capteurs.

1.2.3 Proposition d'un NoC dynamiquement adaptable

Après un état de l'art des solutions de réseau sur puce pour la vision, ce troisième chapitre présente notre nouvelle proposition de réseau appliqué dans un contexte de vision embarquée multi-capteurs et multi-applications. Ce nouveau modèle de réseau, basé sur une solution hiérarchique de routeurs de paquets de données, est capable de s'adapter dynamiquement en terme de chemin de données afin d'exécuter de manière efficace différentes applications. Cette adaptation met en oeuvre des séquencements spécifiques d'opérations de traitement d'image pour une exécution séquentielle et/ou parallèle. Ce réseau met alors en oeuvre une méthode originale d'aiguillage de paquets permettant de traiter plusieurs flux de données en parallèle.

1.2.4 Conception d'un routeur multi-flux dynamiquement adaptable

Le quatrième chapitre étudie et propose une nouvelle architecture de routeur adaptée au modèle de réseau de communication présenté dans la partie précédente. Ce routeur disposant de plusieurs canaux de communication en parallèle, possède la capacité d'aiguiller plusieurs paquets en parallèle en adaptant dynamiquement son chemin de données. Les commandes d'adaptation sont envoyées directement dans l'en-tête des paquets de données. Ces en-têtes comportent des informations sur l'adressage du paquet (source-destination), sur les caractéristiques de la trame image appelées attributs, et des instructions spécifiques correpondant à un séquencement d'opérations de traitement d'image à appliquer sur la donnée pixélique.

1.2.5 Système de mémorisation dynamiquement adaptable

Le cinquième chapitre de ce manuscrit traite du problème de mémorisation des trames. Une solution basée sur un partage statique de la mémoire est insuffisante pour exploiter efficacement notre réseau de communication. Nous voyons que le stockage de trames dans notre contexte nécessite la mise en oeuvre d'une couche d'abstraction pour les accès mémoires permettant de modifier dynamiquement et efficacement une application. Nous proposons ainsi une nouvelle solution de gestion avec une méthode d'accès indirecte aux trames d'image afin de faciliter les changements d'application. Basé sur notre modèle de réseau, nous proposons une organisation mémoire hiérarchique respectant les types d'accès utiles en traitement d'image au niveau pixel, ligne et trame. Nous étudions également, dans ce chapitre, la conception de routeurs spécifiques dotés de la capacité d'accéder à un stockage mémoire partagé de trames.

1.2.6 Validation expérimentale

Le dernier chapitre de ce manuscrit présente une proposition d'architecture du réseau, associé à son système mémoire, dans le but de valider son fonctionnement. Cette architecture, baptisée MDFR (*Multi Data Flow Ring*), est une solution comportant plusieurs unités de calculs agencées dans un réseau de communication avec une topologie en anneau. Nous voyons que ce choix topologique est particulièrement efficace pour des applications orientées flot de données. Les performances en temps et en surface, ont été évaluées sur une cible FPGA avec des applications de visualisation type pour un système de vision portable du CE-CTP Sagem.

Chapitre 2

Système de vision embarqué portable et multi-capteurs

2.1 Introduction

Un système de vision embarqué est un système doté de la capacité de traiter et de visualiser une ou plusieurs sources d'image. Il est soumis à des contraintes d'encombrement et de consommation d'énergie. On peut citer des équipements de type embarqué comme des caméras thermiques de taille imposante utilisées sur des bateaux, ou un système multi-capteurs embarqué dans un véhicule automobile. Le système est qualifié de portable selon le volume et le poids de l'équipement de vision. Pour des équipements portables, nous pouvons citer en exemples des jumelles portables multi-fonctions ou encore "ultra-portable" tel un équipement de téléphonie mobile, embarquant un capteur d'image, tenant dans une simple poche de pantalon.

Malgré cette diversité de types de profil pour l'embarqué, on peut décomposer tout système de vision en trois parties inter-dépendantes, comme illustré par la figure 2.1 : l'*acquisition* d'image contenant les différentes sources d'image, la partie *traitement* des images contenant le coeur électronique de calcul embarqué capable de traiter les différents flux de pixels et la partie *affichage* composée des différents afficheurs en sortie.

L'électronique de calcul embarqué constitue la "clef de voûte" de l'architecture complète du système de vision et sa conception doit ainsi être réalisée avec le plus grand soin. Elle

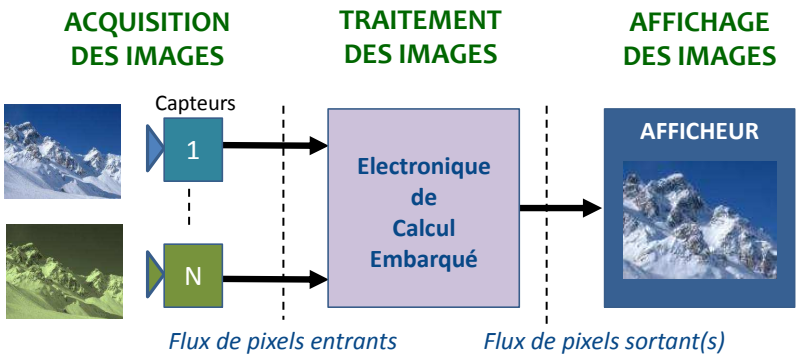


FIGURE 2.1: Diagramme d'un système de vision embarqué avec N capteurs

doit en effet être dimensionnée en fonction des applications à déployer, des différents capteurs et des différents afficheurs, pour assurer des performances en temps tout en respectant les contraintes de l'embarqué en terme d'encombrement et de consommation.

Dans ce chapitre, nous présenterons différents défis liés à la conception de l'électronique embarquée de traitement pour un équipement de vision. Un premier défi concerne la variété de capteurs d'image utilisés dans un même système. Le second défi concerne la variété d'application nécessitant d'être déployée dans un seul système. Ces applications couvrent des opérations primaires de correction d'image provenant des capteurs, vers des opérations avancées impliquant plusieurs sources d'image. Ces opérations sont réalisées à la fois dans le domaine spatial et temporel. Le troisième défi concerne la capacité de l'architecture de calcul à respecter les contraintes de performance du système.

2.2 Acquisition des images

Un système moderne interface plusieurs types de capteurs d'image bien souvent hétérogènes. Ces capteurs ne produisent pas forcément une image parfaite et exploitable et nécessitent pour la plupart une propre chaîne de traitement de correction. De manière générale, les opérations requises, traitent des problèmes liés à l'optique en corrigeant les distorsions par exemple, des problèmes liés à la fabrication du capteur en corrigeant les non-uniformités des valeurs pixéliques (*Non Uniformity Correction*) ou encore des défauts propre à la technologie utilisée comme le bruit dans l'image. Par ailleurs, l'image brute issue d'un capteur couleur à technologie CMOS (*Complementary Metal Oxide Semiconductor*) est inexploitable sans appliquer un traitement de restauration d'image permettant de restituer l'image couleur sous les trois plans, rouge, vert et bleu.

Afin de mieux comprendre la problématique de la variété de capteurs, nous établissons dans cette section une courte présentation de trois types de technologies incontournables dans le domaine de la vision embarquée en condition de jour et de nuit : capteur d'image *couleur*, capteur *bas niveau de lumière* et capteur *infrarouge*.

2.2.1 Capteur d'image couleur

Le capteur couleur est le capteur le plus répandu dans les équipements de vision embarquée portable et plus particulièrement dans le domaine de la photographie numérique [1]. Un capteur numérique réalise l'acquisition et l'échantillonnage d'une image en deux dimensions. Il est constitué d'une dalle de photorécepteurs qui transforment par effet photoélectrique, un signal lumineux de photons en un signal électrique analogique constitué d'électrons. Ce signal est ensuite échantilloné au travers d'un convertisseur analogique/numérique afin de produire un flux de pixels correspondant à la représentation matricielle de l'image.

Il existe deux principales technologies dans le domaine de l'imagerie couleur : la technologie CCD (*Charge-Coupled Device*) et la technologie CMOS (*Complementary Metal Oxide Semiconductor*).

La technologie CCD [21] correspond à une dalle de photorécepteurs qui sont adressés de manière séquentielle. Ce type d'accès est particulièrement lent pour accéder à la valeur d'un pixel. Le convertisseur ADC (analogique/numérique) ainsi que l'amplificateur de gain sont déportés sur un *PCB* (*Printed Circuit Board*) en sortie du capteur.

La technologie CMOS [22] utilisent le principe de pixel actif (APS : *Active Pixel Sensor*). Dans cette technologie, chaque photosite du capteur contient à la fois un photorécepteur, une diode de lecture et un amplificateur, comme illustré par la figure 2.2. Ceci présente ainsi un avantage important par rapport à la technologie CCD concernant la facilité d'accès aléatoire sur un pixel de l'image. Les photosites sont ainsi gérés de façon indépendante grâce à une matrice de commutation. La technologie CMOS apporte donc une véritable souplesse d'utilisation du capteur pour des applications de traitement d'image travaillant sur des zones par exemple. Un autre avantage concerne la compatibilité de cette technologie avec la conception VLSI (*Very Large Scale Integration*) permettant

ainsi d'intégrer des circuits spécifiques, comme par exemple un oscillateur, qui nécessitent usuellement d'être placés dans un PCB déporté du capteur dans le cas de la technologie CCD. Etant donné que chaque pixel possède son propre circuit d'amplification, le facteur de remplissage (*fill factor*) d'un capteur CMOS s'en trouve cependant réduit. La capacité d'intégration du CMOS implique donc une réduction de la taille du capteur, une réduction de la consommation et un coût de fabrication plus faible [23].

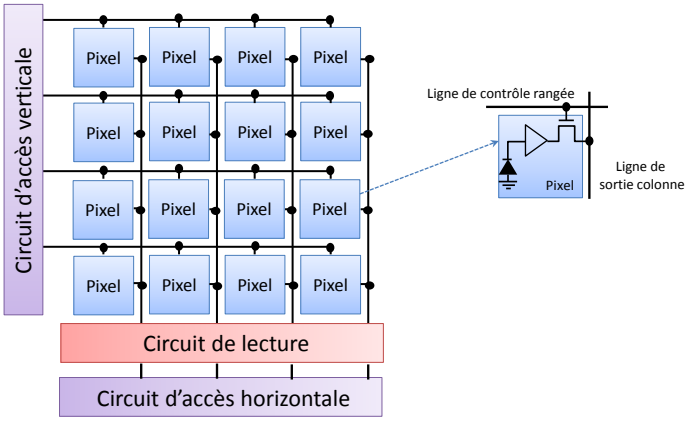


FIGURE 2.2: Architecture d'un capteur CMOS [22]

Un capteur qu'il soit CCD ou CMOS est monochromatique. Afin de pouvoir récupérer une image en couleur, un filtre coloré est appliqué sur la surface photosensible. Ce filtre est une mosaïque spatiale des couleurs primaires rouge, vert et bleu, dont la plus répandue est celle de Bayer [24] introduite par la société Kodak. Dans ce filtre, illustré par la figure 2.3, il y a deux fois plus d'information pour la couleur verte (50 % vert, 25 % rouge et 25 % bleu) en raison de la sensibilité de l'oeil humain à des longueurs d'onde proches du vert.

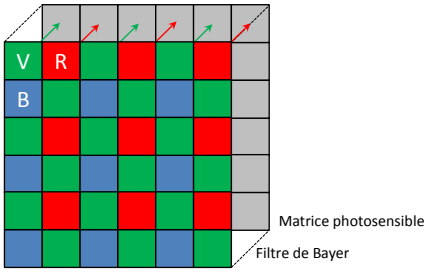


FIGURE 2.3: Filtre de Bayer appliqué à une matrice photosensible

Ainsi, la qualité de l'image finale dépendra fortement de la méthode de reconstruction de l'image, appelée dématriçage (ou démosaïquage), à partir de ces trois plans partiels des couleurs primaires [25].

Un exemple de chaîne de traitement simplifiée pour un capteur couleur est présenté en figure 2.4, afin d'obtenir une image visualisable en sortie sur un afficheur.

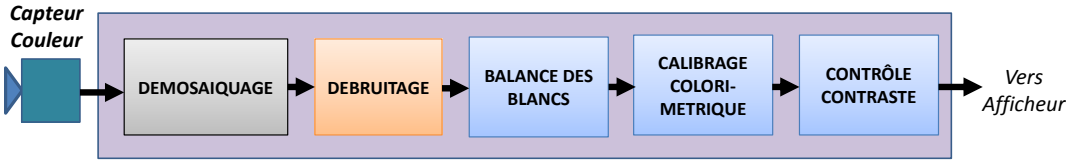


FIGURE 2.4: Exemple de chaîne de traitement pour un capteur couleur

Après l'étape de démosaïquage, l'image peut être améliorée en corrigeant le bruit introduit selon la technologie du capteur. Cette image requiert par la suite une série d'opérations afin d'améliorer sa restitution sur un afficheur. Elle subit ainsi une compensation colorimétrique avec une correction de la balance des blancs. Cette correction consiste à corriger les différences de sensibilité entre les trois canaux rouge, vert et bleu. Le contraste de l'image peut être ensuite corrigé en modifiant par exemple son histogramme. Par définition, l'histogramme est, pour une seule composante de l'image, une fonction discrète qui associe à chaque valeur d'intensité le nombre de pixels prenant cette valeur. Par ailleurs, cette technique de correction par histogramme permet également de cadrer la dynamique de l'image entrante avec la dynamique de l'image de sortie sur un afficheur.

2.2.2 Capteur d'image à bas niveau de lumière (BNL)

Dans des conditions nocturnes, les capteurs couleurs deviennent insuffisants pour l'observation d'une scène. Si les conditions de luminosité sont faibles, il est possible d'utiliser des capteurs spécifiques dits "à bas niveau de lumière" (BNL). Pour ces types de capteurs, une solution est d'amplifier l'information lumineuse transportée par les photons. Cette amplification peut être réalisée par des technologies à base de tube à intensification de lumière (IL). Le principe consiste à appliquer une tension importante dans le tube contenant la photocathode afin d'accélérer les photoélectrons et amplifier le signal en les faisant traverser une galette de micro-canaux (microchannel plate, MCP). Ces tubes peuvent ainsi être utilisé pour réaliser des capteurs en les combinant avec un capteur CMOS (IL-CMOS) ou CCD (IL-CCD). Une solution alternative plus compacte est proposée par la technologie EBCMOS (Electron Bombarded CMOS) [26] qui réalisent directement le bombardement photoélectronique sur le CMOS, comme illustré par la figure 2.5.

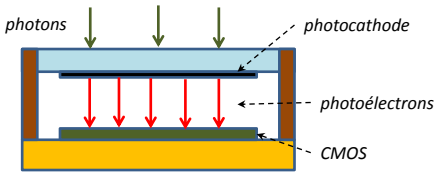


FIGURE 2.5: Capteur Bas Niveau de Lumière de type EBCMOS

Un exemple de chaîne de traitement simplifiée pour un capteur BNL est présenté en figure 2.6, afin d'obtenir une image visualisable en sortie sur un afficheur.

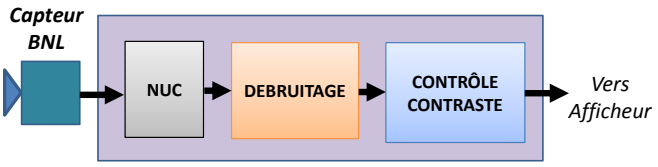


FIGURE 2.6: Exemple de chaîne de traitement pour un capteur BNL

L'image brute en sortie du capteur nécessite dans un premier temps, une correction des non-uniformités (NUC) provenant de la disparité des réponses des photorécepteurs. L'amplification importante de l'information lumineuse dans ce capteur, introduit un bruit dans l'image, propre à la technologie, à minimiser pour obtenir une image exploitable. Comme pour le capteur couleur, l'image subit par la suite un réhaussement de contraste avant d'être affichée.

2.2.3 Capteur d'image infrarouge (IR)

Dans des conditions de noir total, l'amplification de l'information lumineuse devient inutile et il est nécessaire de travailler sur d'autres bandes spectrales. La technologie infrarouge [27] permet de travailler sur des bandes spectrales de 3 à 15 μm ce qui permet ainsi de visualiser une scène en faible luminosité. On distingue deux familles de capteur infrarouge : la famille refroidie et non-refroidie. Dans le cas du refroidi, les capteurs sont des semi-conducteurs à base d'indium et d'antimoine (InSb) qui sont placés à l'intérieur de cryostats à une température inférieure à 80K. Cette technologie propose un très bon rendement quantique mais possède un inconvénient dans la fabrication des caméras qui sont d'encombrement important. Ainsi, cette famille est réservée seulement pour de l'embarqué. La technologie non-refroidie [28] autorise des systèmes plus compacts et se base sur des détecteurs bolométriques dont la conductivité change en fonction de la température. Cette famille de détecteur, dont la température peut être stabilisée par effet Pelletier, introduit néanmoins de nouveaux défis de traitement d'image en sortie du

capteur car sa réponse n'est pas uniforme sur sa matrice de pixels, et est plus sensible au bruit.

Un exemple de chaîne de traitement simplifiée pour un capteur IR est présenté en figure 2.7, afin d'obtenir une image visualisable en sortie sur un afficheur.

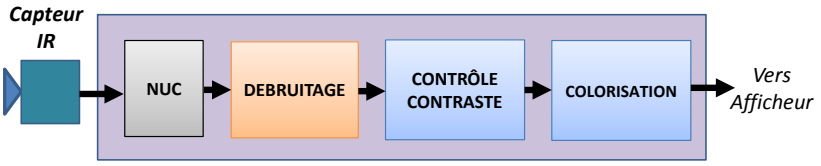


FIGURE 2.7: Exemple de chaîne de traitement pour un capteur IR

Nous pouvons remarquer que la chaîne de traitement d'un capteur IR est très proche du capteur BNL avec un séquencement similaire des opérations à appliquer sur l'image. Hormis la possibilité de colorier une image IR selon une table de couleur indexée par la luminance, les différences entre une chaîne BNL et une chaîne IR se situent principalement dans les méthodes employées dans les étapes de débruitage et de réhaussement de contraste. Ces méthodes sont plus ou moins complexes selon le type de capteur, qui se distingue selon la résolution de l'image et la sensibilité au bruit par exemple. Les algorithmes de ces méthodes ainsi que les techniques d'implémentation associées, constituent le savoir-faire de chaque fabricant de systèmes de vision embarqués, pour obtenir la meilleure image possible.

2.2.4 Contraintes introduites par la multiplicité de capteurs

Comme souligné précédemment, la technologie d'acquisition d'une image dans un capteur n'est pas parfaite. Des capteurs présentent des défauts comme par exemple des zones d'acquisition de l'image dont les valeurs obtenues ne sont pas uniformes, ou encore la présence de bruit dans l'image, qui est une problématique incontournable en vision embarquée portable. Ce bruit correspond à la variation du signal pour une luminance constante. Il provient de la conversion des photons en électrons et de l'électronique d'acquisition et de numérisation.

Le choix d'utilisation d'un ou plusieurs types de capteurs introduit des contraintes en terme de traitement et de performance dans l'électronique de calcul. Comme présenté dans les chaînes de traitement précédentes en figure 2.4, 2.6 et 2.7, certaines opérations permettent de compenser des défauts technologiques, comme par exemple une opération

de correction des non-uniformités (*Non Uniformity Correction*). D'autres traitements sont, par contre, obligatoires, comme par exemple une opération de dématriçage (ou démosaïquage) pour les capteurs utilisant un filtre couleur.

Le tableau 2.1 présente des exemples de capteurs BNL, IR et couleur avec les performances nécessaire pour l'architecture de calcul en terme de bande passante et de fréquence de traitement pixélique. Ce tableau résume pour chaque type de capteur, la taille du pixel en bits, la résolution de l'image avec la fréquence d'acquisition en Hz, le volume de données en mégapixel (Mpix), le débit en mégabit par seconde (Mbps) et la fréquence d'horloge nécessaire pour traiter le flux pixélique (Hpix). Ces performances sont également calculées dans le cadre d'un capteur en haute définition, dit *Full HD* ou *1080p*, avec une résolution de 1920×1080 pixels.

Nous pouvons remarquer que les capteurs utilisés dans le domaine de la photographie numérique possèdent des résolutions d'image imposant un volume de données supérieur à la dizaine de mégapixels [29]. Dans ce tableau, nous nous concentrons principalement sur les résolutions de capteurs utilisées dans un contexte d'acquisition de séquences d'images à une fréquence définie.

Capteurs	Taille pixel (bits)	Résolution	Mpix	Mbps	Hpix (MHz)
BNL	10	1280×1024 (SXGA) @ 50 Hz	1,3	655	65
	12	1280×1024 (SXGA) @ 50 Hz	1,6	786	65
	12	1920×1080 (1080p) @ 60 Hz	2	1500	124
IR	14	640×480 (VGA) @ 25 Hz	0,3	107	7,6
	14	640×480 (VGA) @ 100 Hz	0,3	430	31
	16	1920×1080 (1080p) @ 60 Hz	2	1990	124
COULEUR	24	800×600 (SVGA) @ 25 Hz	0,48	288	12
	24	1280×720 (720p) @ 25 Hz	0,92	553	23
	24	1920×1080 (1080p) @ 60 Hz	2	2488	124

TABLE 2.1: Exemples de caractéristiques de capteurs BNL, IR, et couleur

Nous observons ainsi que la croissance technologique des capteurs en terme de résolution d'image et de fréquence trame, imposent des cadences de traitement de plus en plus élevés de l'ordre du Gbit avec une fréquence d'horloge de pixel dépassant la centaine de MHz.

De plus, cette croissance technologique est couplée avec des besoins applicatifs qui imposent la multiplicité des capteurs dans un même système de vision. Cette multiplicité peut être *homogène*, afin d'obtenir une complémentarité en terme de position et de champ de vision, ou *hétérogènes*, dans le but d'obtenir une complémentarité en terme de sensibilité du capteur.

En effet, pour une utilisation du système de vision dans toutes les conditions de luminosité, l'intégration d'un capteur unique devient insuffisant. Un système de vision moderne est contraint de combiner différentes sensibilité de capteurs afin de pouvoir afficher une image exploitable. Les besoins applicatifs actuels ne se contentent plus d'un simple basculement d'image entre deux capteurs ou d'une double visualisation. Ils s'orientent vers des opérations plus avancées qui combinent intelligemment différentes composantes informationnelles de chaque capteur pour produire une image unique fusionnée [30, 31].

Nous présentons ainsi dans la section suivante, des exemples de besoins applicatifs incontournables dans un équipement de vision embarquée portable.

2.3 Applications dans un équipement optronique

Plus généralement, les applications destinées à la visualisation des images sur un afficheur utilisent un certain nombre d'opérateurs de traitement d'image que nous pouvons classer dans les catégories suivantes : la *restauration*, l'*analyse*, l'*amélioration* et la *restitution* de l'image.

Un exemple d'enchaînement classique de chaque catégorie d'opérateurs est illustré par la figure 2.8 dans le cadre d'une application de visualisation d'une image fusionnée provenant de plusieurs capteurs.

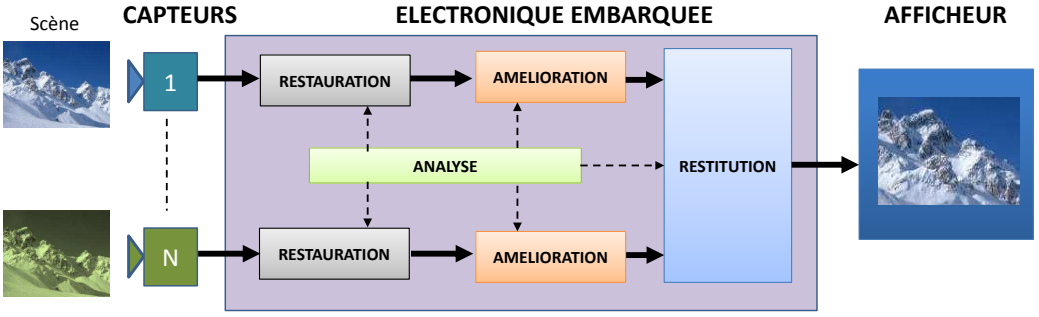


FIGURE 2.8: Application permettant la visualisation d'une image fusionnée de N capteurs

Les opérateurs de restauration permettent de corriger les défauts technologiques du capteur, comme la non-uniformité des pixels, et si besoin, de reconstruire une image, ce qui est le cas pour les capteurs utilisant un filtre couleur. Les opérateurs d'amélioration permettent d'augmenter la qualité de l'image. La visualisation d'une image peut ainsi être améliorée en minimisant le bruit, en optimisant le contraste et si besoin, en

modifiant successivement l'orientation des images pour stabiliser une séquence d'acquisition. Cette visualisation peut aussi être personnalisée par des opérateurs dédiés à la restitution d'image. Ces opérateurs sont utiles pour modifier simplement la colorimétrie, la luminosité de l'image ou encore transformer géométriquement l'image pour agrandir par exemple une zone d'intérêt. Certains opérateurs nécessitent des opérations d'analyse qui fournissent des informations statistiques de l'image, comme un histogramme ou des mesures de mouvement dans une séquence d'images.

Dans la figure 2.8, l'application permettant de visualiser une image fusionnée commence par restaurer et améliorer l'image de chaque capteur puis se termine en combinant ces images pour obtenir une image unique vers un afficheur. L'image de sortie peut ainsi être une image mono-capteur ou une image fusionnée multi-capteurs. Une image fusionnée est le résultat d'une simple incrustation de l'image d'un capteur dans une autre image, ou une combinaison de l'image complète pixel à pixel. Etant donné que les résolutions des images sont différentes, cette dernière opération nécessite une transformation géométrique des images afin de faire correspondre parfaitement les pixels de chaque capteur.

Nous pouvons également observer dans la figure 2.8 que les opérations d'analyse de l'image sont présentes tout au long de l'application afin de calculer des statistiques de l'image, indiquées sous forme de flèches en pointillées dans la figure. Ces informations sont nécessaires pour l'exécution de certaines opérations de restauration, d'amélioration et de restitution.

Une application de visualisation peut être étendue pour réaliser une application de construction de panoramique à partir d'un ou plusieurs capteurs. Cette application permet ainsi d'augmenter le champ de vision du système en projetant plusieurs images dans un repère absolu.

L'image produite par une application de visualisation d'image est utilisable pour d'autres applications de plus haut niveau comme de l'aide à la détection d'objets mobiles dans le but de décamoufler des objets mobiles dans un fond texturé, comme le système présenté dans [32].

Dans le domaine de la sécurité, une image claire et contrastée facilite des applications d'identification comme l'analyse de visage [33, 34] ou l'identification de plaque d'immatriculation comme illustré en figure 2.9.

FIGURE 2.9: Exemple d'application pour l'aide à l'identification de plaque d'immatriculation

Il est important de noter qu'une application de visualisation est variable au cours du temps en fonction du capteur d'image utilisé et en fonction de modes d'utilisation définis pour le système de vision. Les exigences, en terme de latence et de temps de traitement, d'un équipement de vision portable moderne sont ainsi variables au cours du temps, en fonction des modes d'utilisation.

Afin d'illustrer ce propos, prenons l'exemple d'un équipement de vision embarqué portable muni d'un mode de fonctionnement en observation, en déplacement, en capture d'image et en mode veille.

Chaque mode de cet équipement possède une exigence spécifique en terme de latence trame, de qualité d'image et de consommation. Dans un mode d'observation ou de capture d'images, l'équipement devrait mettre plus d'effort en terme de qualité de l'image contrairement à un mode de déplacement du système où une latence la plus faible possible devrait primer. Un mode veille nécessitera, par souci de consommation, de désactiver toutes les fonctionalités optionnelles, en maintenant uniquement les opérations indispensables sur l'image comme ceux de restauration. Cette caractéristique se distingue donc d'une simple caméra de surveillance dont l'application de visualisation consiste seulement à fournir la meilleur image possible avec le souci permanent de performance en temps. Ainsi, la notion de performance doit être relativisée en fonction de l'utilisation.

Afin d'évaluer les besoins en terme de calcul et de mémorisation pour différentes applications, la section suivante présente quelques exemples d'opérateurs de traitement d'image fréquemment utilisés dans un système de vision embarqué portable multi-capteurs.

2.3.1 Analyse de l'image

Nous classons dans la catégorie *analyse* de l'image pour la restitution, toutes les opérations qui calculent des informations statistiques de l'image : moyenne des valeurs des

pixels d'une zone, détection et estimation de mouvement dans l'image.

2.3.1.1 Relevé statistiques

Une image entrante peut être analysée afin de calculer des informations statistiques sur les valeurs de pixels. Cette analyse s'effectue en définissant une ou plusieurs zones d'intérêts dans l'image. Ces statistiques peuvent être, par exemple, la moyenne des valeurs des pixels, la valeur maximum ou minimum, ou encore le nombre de pixels dont la valeur dépasse un seuil fixé pour une zone de l'image définie. La problématique repose alors dans les choix de la zone d'analyse. En effet, une analyse de l'image complète permet d'obtenir un bon rendu global de l'image sans pour autant avoir une bonne qualité au centre où se situe à priori la zone d'intérêt.

Des relevés statistiques de valeurs de pixels dans une image peuvent par exemple être utilisés pour corriger automatiquement de façon pertinente le contraste et la luminance de cette image.

La méthode la plus simple consiste à effectuer un parcours linéaire des valeurs de pixels dans l'image afin d'extraire des statistiques. D'autres méthodes nécessite le calcul de l'histogramme de l'image pour évaluer la distribution des valeurs des pixels dans l'image. Ainsi, ces méthodes imposent une mémorisation de l'image complète et/ou la mémorisation de l'histogramme de cette image pour pouvoir être appliquées.

La figure 2.10 illustre un exemple de relevé statistique de la moyenne globale des valeurs des pixels pour deux images couleurs avec une estimation de la saturation de l'image en analysant l'histogramme.

Suivant l'analyse de cette distribution, une loi peut être définie afin de qualifier qu'une image est sous-exposée (distribution des pixels dans la zone de faible valeurs) ou surexposée (distribution des pixels dans la zone des hautes valeurs).

2.3.1.2 Détection de mouvement dans l'image

Une opération de détection de mouvement a pour objectif d'évaluer la présence de mouvement dans l'image. Cette présence peut être détectée par une simple différenciation de

(a) moyenne des valeurs de pixel à 60 avec 10 % de valeurs saturées

(b) moyenne des valeurs de pixel à 128 avec 20 % de valeurs saturées

FIGURE 2.10: Relevés statistiques d'une image couleur

valeurs de pixels entre des images successives. Un mouvement est ainsi considéré détecté selon un seuil arbitraire fixé pour la valeur de la différence.

La figure 2.11 illustre un exemple de détection de mouvement sur une image couleur en remplaçant les valeurs des pixels qui ont bougés par la couleur rouge, avec un seuil de de différence de valeur fixé à 60.

(a) Image couleur en entrée

(b) Détection de régions en mouvement marqués en rouge

FIGURE 2.11: Exemple d'application d'une détection de régions en mouvement avec un seuil à 60

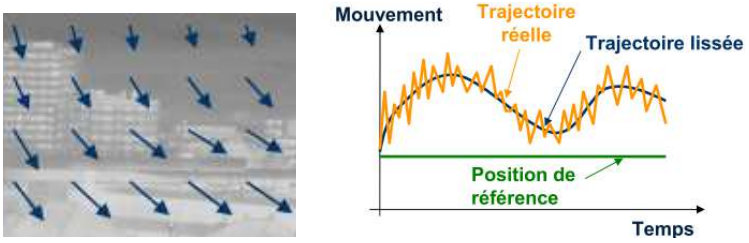
Etant donné qu'un système de vision portable subit des mouvements de l'utilisateur, cette détection doit s'effectuer de manière pertinente. Elle nécessite généralement une stabilisation de l'image afin de recaler les images successives par translation et rotation géométrique. Cette stabilisation ne peut s'effectuer qu'avec une estimation plus fine du mouvement de l'image.

2.3.1.3 Estimation de mouvement dans l'image

La compensation du mouvement apparent dans l'image est une problématique importante dans un système de vision portable. Le mouvement dans l'image provient de deux

sources : le mouvement du système et le mouvement des objets dans la scène observée. Ces mouvements entraînent une perception de filet sur la scène et sur les objets en mouvement.

Une opération d'estimation de mouvement recherche un mouvement d'ensemble simple à partir du mouvement de certains pixels en résolvant l'équation du flot optique [35]. Les valeurs obtenues sont des vecteurs de mouvement, qui peuvent donc être utilisées afin de lisser la trajectoire de l'affichage, comme illustré par la figure 2.12, grâce à des opérateurs de transformation géométrique comme la translation et la rotation de l'image.



(a) Vecteurs de mouvement estimés

(b) Lissage de la trajectoire

FIGURE 2.12: Estimation des vecteurs de mouvement afin de lisser la trajectoire [36]

L'opérateur d'estimation de mouvement est un opérateur coûteux en terme de calculs [35] mais devient indispensable pour des opérateurs temporels travaillant sur des images successives.

2.3.2 Restauration d'image

Nous classons dans la catégorie *restauration* d'image, les opérateurs de traitement capables de corriger les défauts technologiques du capteur ou capables de reconstruire une image provenant d'un capteur utilisant un filtre couleur.

2.3.2.1 Correction des non-uniformités

Ces opérateurs permettent, dans un premier temps, de corriger le bruit spatial fixe dans l'image que nous appelons les *non-uniformités*. Ces non-uniformités proviennent de la disparité des réponses des différents photorécepteurs dans la matrice. Les images en sortie du capteur sont alors inexploitables sans correction. La figure 2.13 illustre un exemple d'image brute en sortie de capteur sans correction des non-uniformités.

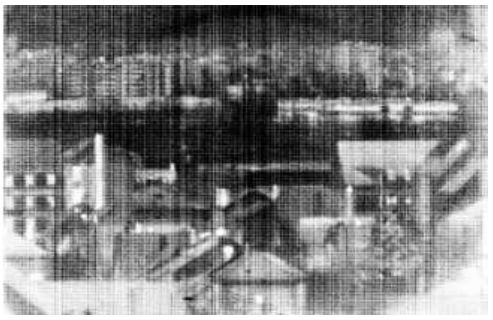


FIGURE 2.13: Image infrarouge sans correction des non-uniformités [36]

Cette correction travaille essentiellement sur deux paramètres de la réponse du pixel, qui est globalement linéaire : le *gain* (correction de la pente) et l'*offset* (ajustement de l'origine). La restauration est ainsi réalisée généralement par la méthode de correction dite "corrections deux points" [37]. Cette méthode utilise une opération consistant à appliquer des tables de correspondances spécifiques $[G, O]$ de gain et d'offset sur les pixels de l'image en fonction du capteur. Soient $G(x)$ le gain et $O(x)$ l'offset à appliquer sur la valeur x du pixel d'entrée de l'image. La valeur x' du pixel de sortie, sera alors :

$$x' = G(x) \times x + O(x) \tag{2.1}$$

Malgré cette correction dans les capteurs infrarouges, il reste bien souvent un résidu de non-uniformité appelé *bruit spatial fixe résiduel* (BSFR), qui augmente avec la distance aux températures ayant servi à créer les tables $[G, O]$. Cette variation du BSFR en fonction du temps d'intégration et de la température du capteur, implique de réaliser plusieurs tables.

2.3.2.2 Dématriçage

Dans le cas de l'utilisation d'un capteur utilisant un filtre couleur, il est nécessaire d'effectuer une opération dite de "dématriçage" (ou "démoisaïquage). La figure 2.14 illustre le résultat de reconstruction des trois plans couleurs à partir d'une image brute, correspondant à un plan partiel pour chaque composante rouge, vert et bleu. Cet exemple illustre l'utilisation d'un filtre de Bayer [24] pour l'image d'entrée avec la prédominance des valeurs pour la composante couleur verte.

(a) Image d'entrée brute (b) Image reconstruite

FIGURE 2.14: Reconstruction d'une image couleur [25]

Il existe différentes techniques dans la littérature [25] : méthode bilinéaire, méthode de Hibbard, méthode de Nakamura ou encore le GEDI (*Green Edge Direction Interpolation*) proposé récemment par Phelippeau [38] pour une application en téléphonie mobile. Pour une utilisation en embarquée, ces méthodes utilisent généralement des opérations de calcul d'interpolation au niveau pixel, permettant ainsi une implémentation en flot de données [38].

Les opérateurs de restauration d'image constituent une première étape essentielle pour obtenir une image exploitable. L'image peut être améliorée par d'autres opérateurs plus complexes afin d'obtenir un résultat plus présentable.

2.3.3 Amélioration de l'image

Nous classons dans la catégorie *amélioration* de l'image les opérateurs se rattachant au problématiques suivantes : la réduction du bruit, la déconvolution d'une image et l'optimisation du contraste.

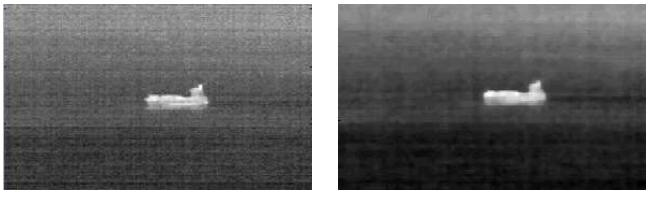
Comme mentionné précédemment, le bruit dans l'image provient de la non-uniformité des réponses dans la matrice. Il provient aussi de la conversion des photons en électrons, appelé bruit quantique, et du circuit électronique d'amplification et de numérisation du signal analogique. On observe ainsi dans le cas d'une scène fixe, la présence de fourmillement dans l'image comme illustré en figure 2.15 dans le cas d'un capteur BNL.

FIGURE 2.15: Exemple d'une image brute non débruitée issue d'un capteur BNL

La littérature autour de la problématique de bruit dans l'image est considérable [39] et différentes méthodes ont été proposées avec une complexité calculatoire variée. On peut classer les opérateurs de débruitage en deux catégories : les opérateurs spatiaux et les opérateurs temporels.

2.3.3.1 Réduction du bruit par méthode spatiale

La réduction spatiale du bruit est une approche classique consistant à remplacer la valeur d'un pixel en fonction de la valeur des pixels dans un voisinage de taille et de forme définies. La valeur de ce pixel peut être, par exemple, une moyenne ou la médiane des valeurs des pixels dans un voisinage donné [8]. Dans ce dernier cas, le filtre médian atténue le bruit mais engendre une perte en terme de contraste sur les bords des formes et les zones texturées, comme illustré par la figure 2.16. On peut citer également d'autres méthodes comme le filtre bilatéral [40] travaillant à la fois en fonction des poids spatiaux et des poids de différences d'intensité avec le pixel origine.

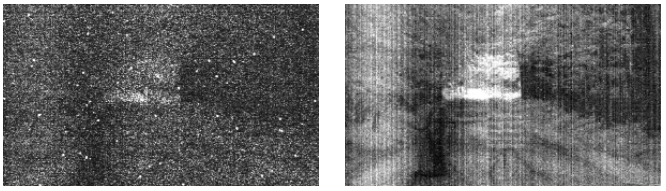


(a) Image IR bruitée (b) Image IR filtrée en median

FIGURE 2.16: Exemple d'application d'un filtre median 5x5 sur une image infrarouge [36]

2.3.3.2 Réduction du bruit par méthode temporelle

Les techniques temporelles de réduction du bruit, répondent à la problématique de bruit additif, et consistent à effectuer une moyenne des valeurs de pixels en accumulant plusieurs images successives dans le temps [8]. La figure 2.17 illustre un résultat obtenu par accumulation d'images dans le cas d'un capteur BNL.



(a) Image BNL bruitée (b) Image BNL débruitée

FIGURE 2.17: Exemple d'application d'un filtre temporel par accumulation d'images BNL [36]

En considérant que le bruit est une variable aléatoire q de moyenne nulle, soient (q_m) une suite de n réalisations de q avec $1 \leq m \leq n$, et (f_m) une suite d'image bruitée par (q_m). Dans le cas de l'hypothèse d'une image sans mouvement, nous avons : $f_m = g + q_m$ avec g étant la composante de l'image requise sans bruit. Le principe consiste alors à effectuer une moyenne sur les images (équation 2.2).

$$\left(\frac{1}{n}\right) f_m = \left(\frac{1}{n}\right) \sum_{m=1}^{n} (g + q_m) = g + \left(\frac{1}{n}\right) \sum_{m=1}^{n} q_m \tag{2.2}$$

Ainsi, pour une valeur de n importante, le signal de sortie est théoriquement sans bruit.

L'algorithme de cette méthode implique la mémorisation de plusieurs trames pour appliquer une moyenne temporelle des valeurs, pixel à pixel. Afin de minimiser le stockage des trames, une méthode récursive peut être utilisée. Soient T_k la trame à l'instant k, I_k l'image intégrée à l'instant k, $NbInt$ le nombre d'intégration et p le pixel traité. Le pixel de l'image $I_k(p)$ sera ainsi calculé par la formule récursive suivante :

$$I_k(p) = \frac{T_k(p)}{NbInt + 1} + \frac{NbInt \times I_{k-1}(p)}{NbInt + 1} \tag{2.3}$$

De manière évidente, cet opérateur de débruitage temporel n'est pas pertinente avec la présence d'objets mobiles dans la scène. Elle implique ainsi l'utilisation des résultats

d'un opérateur de détection de mouvement dans l'image pour désactiver l'opérateur. Il reste efficace pour une utilisation en observation fixe. Notons également que pour être applicable, dans le cas de mouvement du système de vision portable, cet opérateur nécessite un autre opérateur de transformation géométrique pour le recalage spatiale des images successives.

2.3.3.3 Méthode de déconvolution

Un capteur couplé à son optique est caractérisé par sa *Fonction de Transfert de Modulation* (FTM) correspondant à sa réponse fréquentielle. Une optique a une action de filtre passe-bas sur la scène. Les conséquences d'une FTM imparfaite sont la perte de détail avec des contrastes moins visibles en hautes fréquences et un repliement de spectre si la fréquence de coupure de l'optique est trop importante par rapport au capteur. Ces dégradations peuvent être réduites par une méthode de déconvolution. Cette méthode se base sur le modèle de construction d'un image I (équation 2.4) qui est obtenue par une convolution de la scène s par la fonction d'étalement du point, appelée Point Spread Function (PSF) h, et l'ajout d'un bruit n supposé gaussien.

$$I(x, y) = s * h(x, y) + n(x, y) \tag{2.4}$$

Une opération, fréquemment utilisée pour récupérer les fréquences atténuées, est le *filtre de Wiener* [41] qui, après approximation du rapport du bruit n sur le signal de la scène s en une constante, devient un filtre local facilement implémentable en embarqué (figure 2.18).

(a) Image infrarouge en entrée (b) Image infrarouge déconvoluée

FIGURE 2.18: Exemple d'application d'un filtre de Wiener approximé

Il existe également d'autres méthodes directes plus performantes utilisant les ondelettes [42] ou des méthodes itérative comme la méthode de Richardson [43].

2.3.3.4 Optimisation du contraste de l'image

Le contraste peut être optimisé par des méthodes permettant de révéler des informations dans les zones d'ombres dans l'image, en récupérant des détails par des opérateurs globaux et locaux. Ces méthodes engendrent un rapport contraste sur bruit moins bon dans les zones sombres que dans les zones claires mais permet de ne pas saturer l'image. Le contraste est aussi impacté au moment de la réduction de la dynamique de l'image acquise vers un afficheur. Ainsi, en prenant l'exemple (figure 2.19) d'une image infrarouge acquise en 14 bits par pixel, destinée à être envoyée vers un afficheur 8 bits, il est nécessaire de déterminer une méthode de réduction de la dynamique permettant de préserver le contraste.

(a) Image IR divisée par 64 (b) Image IR en méthode globale (c) Image IR en méthode locale

FIGURE 2.19: Méthodes de réduction de dynamique avec optimisation du contraste [36]

Dans notre exemple présenté en figure 2.19, une simple division (par 64 dans notre exemple) est insuffisante pour conserver les détails.

Une méthode globale par compression de la dynamique est une solution consistant à comprimer chaque valeur de la dynamique en entrée vers une valeur de la dynamique de sortie. En terme de calcul, un opérateur global peut se résumer à une opération arithmétique qui applique un gain global sur l'ensemble des pixels de l'image, contrairement à des méthodes locales.

Les méthodes locales nécessitent d'effectuer un réhaussement de luminance suivant l'analyse du contenu localement dans l'image, ce qui permet d'obtenir un meilleur piqué dans l'image. Cependant, elle ne permettent plus de conserver la photométrie de l'image. Ces méthodes, dont le principe est d'augmenter le contraste entre un point et son voisinage,

sont plus complexes et peuvent nécessiter de combiner plusieurs images à différentes échelles. Nous pouvons citer en exemple l'algorithme *Retinex* [44].

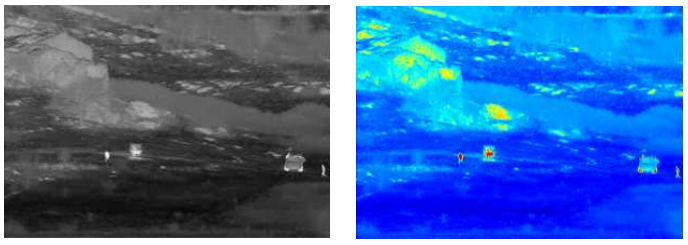
En résumé, les opérateurs dédiés à l'amélioration d'image permettent ainsi d'obtenir une image plus présentable après une étape de restauration. Nous notons que l'ensemble de ces opérateurs agissent de manière autonome sans intervention de l'utilisateur.

2.3.4 Restitution d'image

Les opérateurs de *restitution* d'image sont en interaction avec l'utilisateur. L'image en sortie sur un afficheur externe est généralement personnalisable. Les opérateurs dédiés à la restitution sont activés suivant des besoins ponctuels d'utilisation, afin d'optimiser le rendu final de l'image : ajout de couleurs dans une image monochrome, gestion de la luminosité-contraste, zoom numérique ou encore des techniques de super-résolution.

2.3.4.1 Colorisation de l'image

Cet opérateur s'applique principalement dans le cas des images infrarouges. Une image infrarouge est monochrome avec des niveaux de gris de plus en plus élevés en fonction des zones de chaleur détectées. Le principe de la colorisation de cette image par une méthode dite de *fausses couleurs* [45], consiste à associer à chaque niveau de gris une couleur en utilisant une table de correspondance indexée par la luminance. Il est donc possible de définir plusieurs tables de couleurs selon l'utilisation du système de vision pour mode dédié à la détection ou un mode dédié à la mobilité. La figure 2.20 illustre un exemple d'application d'une table de couleur sur une image infrarouge.



(a) Image IR en entrée (b) Image IR en fausses couleurs

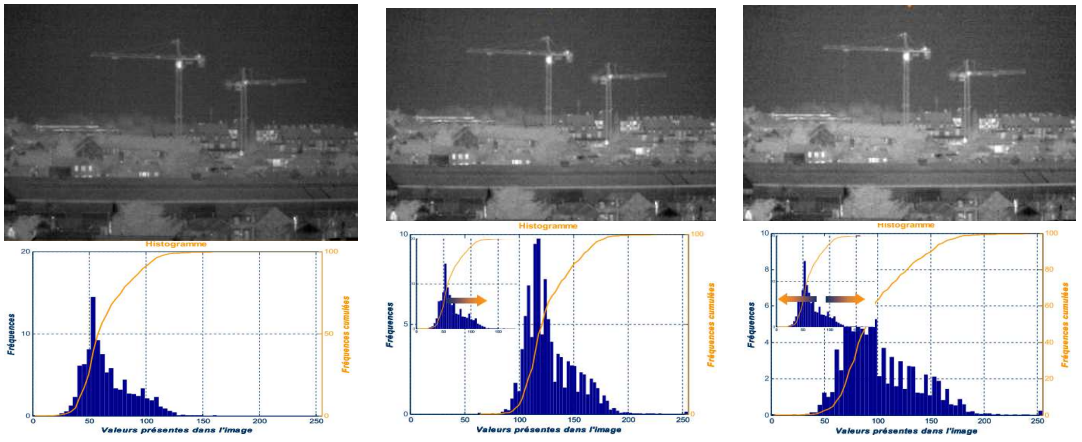
FIGURE 2.20: Opérateur de fausses couleurs sur une image IR

Le rendu final peut être ainsi plus pertinent car l'oeil humain perçoit plus de couleurs que de niveaux de gris.

2.3.4.2 Modification de luminosité et contraste de l'image

Nous avons vu précédemment en section 2.3.3.4, que le contraste est ajusté automatiquement afin d'obtenir l'image la plus présentable possible. Cette optimisation se base sur une statistique effectuée dans l'image qui n'est pas forcément toujours la plus pertinente suivant la zone observée et l'utilisation pour révéler des détails supplémentaires dans l'image.

Afin de régler la luminosité et le contraste d'une image, une méthode simple consiste à travailler directement sur l'histogramme de l'image. Un histogramme décrit la distribution des luminances dans l'image. La luminosité peut alors être modifiée, pour assombrir ou éclaircir l'image, par translation d'histogramme. Le contraste peut aussi être modifié par une compression ou une expansion de l'histogramme. La figure 2.21 illustre un exemple de modification de luminosité et du contraste avec une modification d'histogramme.



(a) Image IR en entrée

(b) Modification de luminosité (translation)

(c) Modification de contraste (expansion)

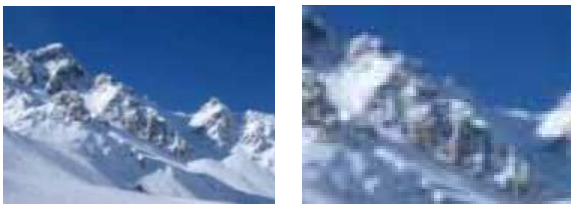
FIGURE 2.21: Réglage de luminosité et contraste dans une image IR

2.3.4.3 Transformation géométrique de l'image

Le choix d'un afficheur en sortie d'un système de vision conditionne également l'intégration de certains opérateurs. Par exemple, des opérateurs sont nécessaires pour comprimer intelligemment la dynamique de l'image d'entrée pour son affichage. D'autres opérateurs sont également utiles pour modifier la résolution spatiale de l'image d'entrée pour s'adapter à celle de l'afficheur.

Dans ce dernier cas, l'image subit des transformations géométriques de manière continue par le biais d'un opérateur d'interpolation. La qualité de l'image transformée est dépendante des méthodes d'interpolation utilisées. Ces méthodes peuvent utiliser une simple opération d'interpolation bilinéaire [46] ou des opérations plus complexes calculant des splines cubiques [47, 48] par exemple.

Cet opérateur peut être également utilisé pour effectuer un zoom numérique de l'image d'entrée. L'opération associée consiste à agrandir une région d'intérêt (ROI) de l'image d'entrée par interpolation. La figure 2.22 illustre un exemple de zoom numérique appliqué sur une image couleur d'entrée.



(a) Image couleur en entrée (b) Image couleur zoomée

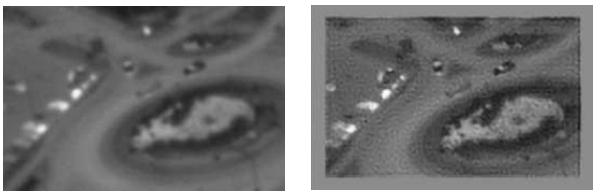
FIGURE 2.22: Zoom numérique pour une image couleur

Nous pouvons remarquer qu'une opération de zoom numérique n'apporte pas plus d'information dans l'image, mais ajoute cependant un confort de visualisation pour l'utilisateur.

2.3.4.4 Super-résolution

L'opérateur de super-résolution [49] est utilisé pour traiter des images de faibles résolutions comme la majorité des capteurs infrarouges. Cet opérateur permet d'augmenter la résolution en exploitant le mouvement dans l'image. L'opération est différente du zoom numérique car elle ajoute de l'information dans l'image. Le principe de cette opération consiste à estimer l'image haute résolution la plus vraisemblable en connaissant la PSF (connue ou estimée), les mouvements et les images basses résolutions.

La technique de super-résolution bénéficie de plusieurs avantages comme la suppression du repliement de spectre 2D, la restauration du flou et une réduction du bruit dans l'image. Nous pouvons citer un exemple récent d'utilisation, comme celui de Chikamatsu [50] pour améliorer la vision en infrarouge avec plusieurs capteurs.



(a) Image IR en entrée (b) Image IR super-résolue

FIGURE 2.23: Super-résolution appliquée à une image IR

2.3.4.5 Fusion multi-spectrale

Afin d'améliorer la perception de la scène, un système de vision doit être capable de produire une image pertinente en tout temps de jour comme de nuit. Une opération de fusion multi-spectrale consiste à produire une image unique à partir de plusieurs sources images provenant de plusieurs capteurs à différentes sensibilités, afin d'exploiter plusieurs bandes spectrales comme le visible, l'infrarouge ou encore le proche infrarouge. L'opération de fusion d'images la plus simple repose sur une combinaison linéaire des valeurs des pixels de chaque image d'entrée pour une même coordonnée. Dans un contexte multi-capteurs hétérogènes, cette opération nécessite, de manière évidente, une opération préalable de transformation géométrique pour chaque image d'entrée, dans le but de les redimensionner et de les harmoniser spatialement. Une fusion pixel à pixel n'est réalisable que si les images sont parfaitement recalées. D'autres méthodes de fusion plus complexes existent [30, 31, 51] et peuvent nécessiter des opérations de pré-traitement spécifiques de l'image comme par exemple une segmentation des images d'entrées [52–54].

La figure 2.24 présente l'exemple d'une image fusionnée, par opérateur pixel à pixel, entre une image d'un capteur monochrome dans la bande spectrale visible avec une autre image dans la bande spectrale infrarouge.

(a) Image visible en entrée (b) Image IR en entrée (c) Image fusionnée et zoomée

FIGURE 2.24: Méthode de fusion d'image visible et infrarouge

Les applications de restitution d'image utilisant cet opérateur sont fréquemment utilisées pour la surveillance d'une zone en tout temps ou pour de l'assistance au pilotage de nuit en mauvaise condition météorologique.

2.3.5 Synthèse

Nous avons présenté une liste non exhaustive de type d'opérateurs de traitement d'image incontournables dans un système de vision embarqué portable.

Le tableau 2.2 effectue un récapitulatif de ces opérateurs pour les différentes catégories : analyse, restauration, amélioration et restitution. Pour chaque opérateur, nous indiquons les besoins de mémorisation qui peuvent être de quelques lignes de l'image d'entrée jusqu'à une image complète. Le stockage d'une image complète est nécessaire pour tous les opérateurs temporels travaillant sur des images successives. Nous rappelons, pour chaque opérateur, l'espace de calcul, qui est soit uniquement spatial sur une image, temporelle sur une ou plusieurs images, ou les deux à la fois. Nous indiquons également le nombre d'images en entrée et en sortie de chaque opérateur, ainsi que leur applicabilité pour un ou plusieurs type de capteurs parmi l'infrarouge (IR), le bas niveau de lumière (BNL) ou la couleur (CL).

Catégorie	Opérateur	Mémoire	Entrée(s)	Sortie	Espace	Capteurs
Analyse	Statistiques	Trame	1	1	S	Tous
	Détection de mouvement	Trame	2	1	T	Tous
	Estimation de mouvement	Trame	2	1	S,T	Tous
Restauration	Dématriçage	Lignes	1	1	S	CL
	NUC	Non	1	1	S	Tous
Amélioration	Débruitage spatial	Lignes	1	1	S	IR,BNL
	Débruitage temporel	Trame	2	1	T	Tous
	Déconvolution	Non	1	1	S	IR
	Contraste global	Non	1	1	S	Tous
	Contraste local	Lignes	1	1	S	Tous
Restitution	Fausses couleurs	Non	1	1	S	IR
	Contrôle histo	Non	1	1	S	Tous
	T.Géométrique	Lignes	1	1	S	Tous
	Super-résolution	Trame	2	1	S,T	IR
	Fusion pixélique	Non	2	1	S	Tous

TABLE 2.2: Tableau récapitulatif des opérateurs pour une application de visualisation

Dans un premier temps, nous remarquons que la majorité des opérateurs effectue des opérations arithmétiques au niveau pixel et autorise un calcul en flot de données.

Nous pouvons également constater qu'en terme de nombre de flux d'entrée et de sortie, la majorité des opérateurs effectuent des calculs sur un seul flux (un flux en entrée et un flux en sortie).

Les opérateurs temporels, comme celui de débruitage, sont multi-flux et effectuent des calculs sur plusieurs trames d'indices temporels différents. Des opérateurs de fusion pixélique de trames sont également multi-flux mais effectuent des calculs sur deux sources d'image différentes.

Par ailleurs, nous remarquons qu'un capteur couleur peut utiliser la majorité des opérateurs cités pour différentes applications. Ces opérateurs ne sont cependant applicables que sur la seule composante de luminance. Ainsi, une source couleur codée en RGB (Rouge, Vert et Bleu) subie généralement un changement d'espace colorimétrique comme le YCrCb ou le HSV (Hue Saturation Value) afin de pouvoir travailler sur la luminance de l'image.

En terme de contraintes de mémorisation, les opérateurs temporels nécessitent naturellement de mémoriser un ou plusieurs trames d'image pour effectuer les calculs. Pour les opérateurs spatiaux, le stockage de la trame peut être nul dans le cas d'application de tables de valeurs indexées par la luminance, comme la table de correction des non-uniformités. Le stockage peut être aussi équivalent à quelques lignes de l'image, afin de travailler sur un voisinage par exemple comme pour un opérateur de débruitage spatial, ou d'accéder aléatoirement sur une partie de l'image, comme pour un opérateur de transformation géométrique de l'image.

Une application de visualisation d'image est spécifiée pour chaque type de capteur d'image en entrée d'un système de vision. Chaque application comporte des opérateurs obligatoires et spécifiques, comme l'opérateur de dématriçage dans le cas d'un capteur couleur, mais nous pouvons remarquer que la majorité des opérateurs sont réutilisables et communs entre les capteurs, comme le réhaussement de contraste ou l'opérateur de transformation géométrique.

Ainsi, les principales modifications d'applications entre les capteurs, se situent sur la configuration de ces opérateurs, avec des coefficients de calcul différents par exemple, et sur l'enchaînement de ces opérateurs. Pour une source capteur donnée, un changement de mode d'utilisation consiste bien souvent à activer ou désactiver un opérateur, et à

changer également le séquencement des opérateurs, dans le but d'accélérer un traitement ou d'améliorer la qualité du traitement de l'image.

Dans ce contexte, concevoir une architecture de calcul unique pour une chaîne de traitement fixe, définie selon un compromis pour l'ensemble des modes d'utilisation, n'est pas la solution la plus adéquate. De plus, dans un contexte de système multi-capteurs, se trouve également posés les problèmes liés à la gestion de différentes granularités de pixels et d'optimisation des accès à la mémoire en fonction de la taille des images.

Nous étudions ainsi, dans la section suivante, différentes architectures de calcul pour l'embarqué, disponibles pour un concepteur de système de vision portable. Nous discutons également de la flexibilité, au niveau architectural, de chaque solution dans le cadre de notre contexte de système de vision multi-capteurs.

2.4 Architectures de calcul pour la vision

Pour les architectures de calcul embarquées, nous nous concentrons principalement sur la *conception au niveau système sur puce* (SoC), lequel est capable de supporter les différents besoins présentés précédemment. Différentes architectures ont été proposées dans le domaine de la vision embarquée portable, tout au long des avancées et de la maturité des technologies d'intégration électronique [55].

Au regard des différentes fonctionnalités nécessitant d'être embarquées dans un équipement, la conception de l'architecture de traitement doit être polyvalente afin de pouvoir effectuer différents types de calculs avec différents modes d'exécution. Ces différentes applications doivent naturellement être déployées de manière performante à la fois en temps et en surface d'utilisation du silicium.

Etant donné la diversité des conditions d'utilisation et de l'environnement en jour ou en nuit, la notion de performance en temps de traitement dans un système de vision devient relative suivant les modes de fonctionnement d'un système (observation, déplacement, veille).

L'intégration de multiples capteurs dans un même système ajoute une couche de complexité supplémentaire à la problématique de conception d'architecture à la fois performante et polyvalente. Afin d'assurer cette polyvalence, par rapport à une architecture

dite câblée, il est nécessaire d'introduire des technologies capables de modifier le calcul selon les applications. Dans ce but, deux technologies se complètent dans le monde de l'embarqué : la technologie programmable et la technologie reconfigurable.

Nous proposons ainsi dans cette section de faire un point sur différents types d'architectures de calcul marquantes de ces dernières années dans le domaine de la vision embarquée.

Dans ce manuscrit, nous utilisons la dénomination de PE pour *Processing Element* afin de définir une unité de calcul élementaire. Nous la considérons comme une "boîte noire" (*black box*) pouvant être un processeur ou une unité de calcul spécifique et optimisée qu'on nomme *Intellectual Property* (IP).

2.4.1 Architectures câblées

Lorsqu'une application de traitement d'image est définie, une première solution consiste à décrire matériellement des unités de calcul optimisées en surface et en temps pour chaque opérateur, sous forme d'IPs, nécessaires à la réalisation de l'application. L'application est ainsi implémentée en assemblant l'ensemble des IPs suivant un séquencement défini.

La structure pipeline correspond à l'assemblage des IPs le plus fréquent dans le domaine de la vision. Un exemple est proposé en figure 2.25 pour un pipeline de 3 IPs : débruitage, réhaussement de contraste et zoom.

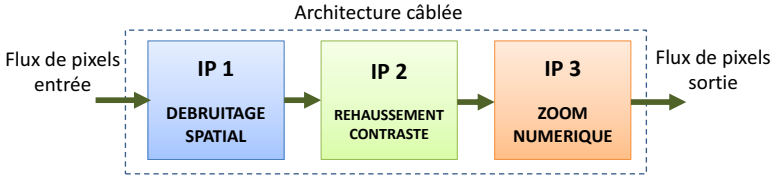


FIGURE 2.25: Structure pipeline d'IPs câblés

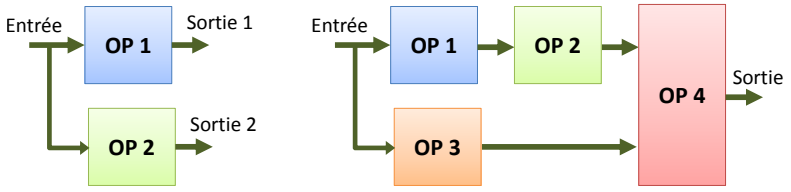
La structure pipeline permet ainsi d'exploiter un parallélisme temporel avec un calcul sur des données différentes. Elle permet également de minimiser le stockage des données avec des IPs capables de traiter directement en flot de données.

Cette méthode reste la plus naturelle dans le domaine de la vision embarquée du fait du flot de données pixélique provenant de la caméra et de la nécessité de tenir une cadence élevée de traitement avec le minimum de latence pour l'affichage. Elle permet de décrire à la fois un parallélisme temporel et spatiale. Le parallélisme spatial peut ainsi être

exploité à l'intérieur d'une unité de calcul au niveau des données, mais aussi entre les unités avec le traitement parallèle de plusieurs flots de données pixéliques.

Un très grand nombre d'architectures câblées ont été proposées en ciblant des opérateurs spécifiques de traitement d'image comme des opérateurs dits bas-niveau [56] ou d'estimation de mouvement [57, 58].

Nous avons constaté dans la section 2.3 précédente que la majorité des opérateurs étudiés dans notre contexte sont particulièrement adaptés à du calcul en flot de données. Chaque opérateur peut ainsi être implémenté de manière optimisée et l'implémentation d'une application consiste à définir l'enchaînement de ces opérateurs. Dans ce cadre, nous pouvons remarquer que la structure pour l'enchaînement de ces opérateurs peut être de trois types : *pipeline*, comme illustré par la figure 2.25 précédente, *parallèle* ou *pipeline-parallèle*, illustrés par la figure 2.26.



(a) Structure parallèle (b) Structure pipeline-parallèle

FIGURE 2.26: Structure parallèle et pipeline-parallèle d'opérateurs câblés

La structure que nous appelons *pipeline-parallèle* est la plus complexe car elle nécessite de combiner plusieurs flot de données pipelinés et exécutés en parallèle. Dans notre étude, cette structure intervient pour les opérateurs de fusion d'image par exemple.

Malgré un temps de développement naturellement important, du fait de la description complète d'une chaîne, les solutions câblées sont une solution idéale pour les systèmes mono-application optimisés avec des contraintes fortes en temps. Néanmoins, suivant la complexité de la chaîne de traitement décrite pour une application, il devient de plus en plus difficile pour un concepteur de faire évoluer cette chaîne sans effectuer des modifications majeures dans les liens de communication mais aussi dans les opérateurs de la chaîne, tout en respectant la dépendance des opérations. Ainsi, dans un contexte où les applications évoluent rapidement, ce travail d'optimisation pour des architectures câblées ne doit être réservé que pour des fonctions critiques et peu évolutives. Nous pouvons citer l'exemple d'un opérateur de transformation géométrique pouvant être

"figé" en connaissant la résolution d'affichage de sortie et la marge de résolution d'image en entrée.

2.4.2 Architectures programmables pour l'embarqué

Nous faisons une distinction entre la notion de reconfigurabilité et la notion de programmabilité. Nous définissons la notion de programmabilité du système au niveau de l'utilisation d'un processeur (programme) et pour la simple "programmation" de multiplexeurs dans une architecture.

Dans la famille des solutions programmables dans le domaine de l'embarqué, nous distinguons les processeurs *généralistes* et les processeurs *spécialisés* embarqués.

2.4.2.1 Processeurs généralistes dans l'embarqué

Un processeur généraliste, aussi appelé *General Purpose Processor* (GPP), autorise la plus grande versatilité en terme de calcul. Nous pouvons citer en exemple les processeurs *MIPS* [59] ou les processeurs *PowerPC* [60] developés conjointement par les sociétés *Apple*, *IBM* et *Freescale* (anciennement *Motorola*).

Dans le domaine de l'embarqué, ce type de processeur est généralement utilisé pour du contrôle global, des applications de haut-niveau comme par exemple un système d'exploitation dédié à l'embarqué et la gestion d'une interface graphique homme-machine (IHM).

Parmi les plus répandus, les processeurs embarqués basés sur la micro-architecture *Bonnell* [61], capable d'exécuter deux instructions par cycle d'horloge, sont des exemples de ces processeurs généralistes optimisés pour la basse consommation en énergie. Ils sont commercialisés sous l'appellation *Atom* par la société Intel. Dans cet exemple, il s'agit d'un architecture dite *In-Order*, dont les instructions sont traitées dans l'ordre d'arrivée, sans ré-ordonnancement. Cette solution permet d'améliorer l'efficacité du pipeline tout en minimisant la taille de la puce. La figure 2.27 illustre l'organisation du chemin de données.

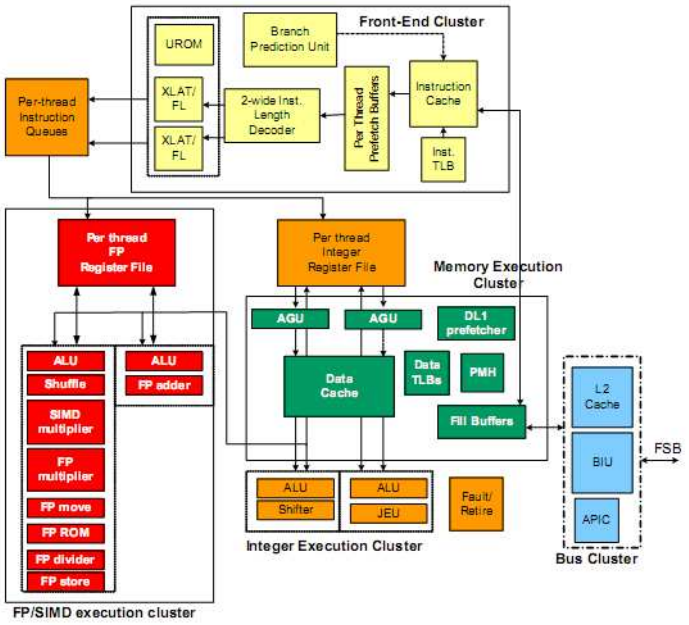


FIGURE 2.27: Chemin de données des processeurs *Atom* (*Intel*) [61]

Notons que cette architecture est dotée d'un pipeline de 16 étages et peut être cadencée à 800 MHz pour une consommation de l'ordre de 3 Watts, avec une surface de 26 mm^2 (gravure à 45 nm).

Ces processeurs dominent le marché actuel des micro-ordinateurs portables, appelés aussi *netbooks*, mais leurs performances restent cependant bien insuffisantes pour des applications complexes et leur consommation trop importante pour des équipements dits ultra-portables, en particulier dans le domaine de la téléphonie mobile.

Dans ce domaine, ce sont les processeurs basés sur les architectures RISC 32-bits de la société *ARM* qui sont prédominants. Ces processeurs sont dotés d'une architecture simplifiée et orientée vers la basse consommation. Basée sur une stratégie de vente sur licence, ces processeurs se sont très rapidement répandues dans diverses propositions de SoC par différents fabricants comme *Texas Instruments*, *Samsung*, *Qualcomm* ou encore *Marvell*. La figure 2.28 illustre un coeur de processeur ARM *Cortex-A9* [62].

Ce processeur est capable d'être cadencé à 830 MHz pour délivrer, sur une surface de 1,5 mm^2 (sans cache), une puissance de calcul de 2075 Dhrystone MIPS (DMIPS) avec une consommation de 0,4 Watts. Il contient également une unité de type SIMD, appelée *NEON*, de taille 128 bits, afin d'accélérer des calculs de type multimédia en exploitant un parallélisme spatial des calculs sur la donnée.

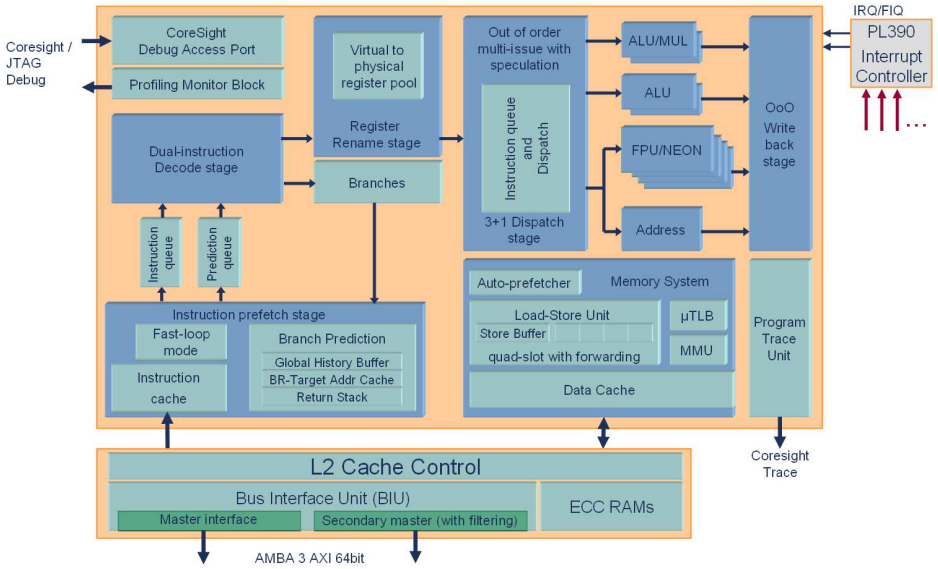


FIGURE 2.28: Architecture du processeur Cortex-A9 (*ARM*) [62]

A travers ces quelques exemples, nous comprenons bien que l'objectif d'un processeur généraliste embarqué est de proposer de la performance en temps et en surface, avec le maximum de flexibilité de calcul, tout en garantissant une consommation électrique réduite. Cependant, dans le domaine de la vision, les unités fonctionnelles généralistes permettent difficilement au processeur d'atteindre une puissance de calcul suffisante pour réaliser une application complète de visualisation avancées, comme présenté précédemment, et cela malgré différentes techniques de parallélisation spatiale et temporelle. Ces processeurs se heurtent en effet à la limite technologique au niveau fréquentiel amenant à contredire la fameuse loi de Moore. La tendance actuelle des processeurs généralistes est de démultiplier les coeurs de traitement de manière spatiale afin de proposer de nouvelle stratégie de parallélisation des calculs. Nous pouvons citer en exemple les processeurs *ARM Cortex A9 MPCore* [63] basés sur le coeur présenté précédemment, qui sont capable de se décliner jusqu'à quatre coeurs de traitement.

Cette montée en nombre de processeurs sur une seule puce, appelée *Multi-Processors SoC* (MPSoC), engendre un surcoût évident en terme de surface silicium, et apporte de nouveaux défis en termes d'efficacité énergétique, de scalabilité en nombre de processeurs et de stratégie de parallélisation pour obtenir un véritable gain en performance [64].

2.4.2.2 Processeurs spécialisés

Lorsque le domaine d'application est bien ciblé, une solution pour augmenter les performances d'un processeur, consiste à concevoir des unités fonctionnelles spécifiques dans le processeur afin d'accélérer les calculs qui sont récurrents dans le domaine. Ainsi, le niveau de spécialisation du processeur est directement lié à la spécificité de ses unités de calcul et à leur disposition spatiale, exploitant un parallélisme de données et un parallélisme temporel avec une disposition de type pipelinée.

Un processeur de traitement de signal, appelé aussi *Digital Signal Processor* (DSP), est typiquement spécialisé, comme son nom l'indique, dans le domaine du traitement de signal, couvrant ainsi des applications en télécommunication, audio ou traitement d'image. Ce type de processeur permet d'être suffisamment flexible pour couvrir un large champ d'applications dans un équipement multimédia portable. Ces processeurs se basent sur l'unité de calcul de multiplication-accumulation (MAC) qui est fréquemment utilisée pour des applications de traitement du signal, comme la compression ou la décompression de flux audio ou pixélique.

Nous pouvons citer en exemple les processeurs DSP proposés par la société *Texas Instruments*. Leurs derniers DSP basé sur le coeur *C66x* sont capables de réaliser des calculs en virgule fixe et en virgule flottante. Un coeur *C66x* (figure 2.29) [65] est ainsi capable d'atteindre une performance de 32 GMACs en fixe sur 16 bits, et 16 GFLOPs en flottant sur 32 bits, pour une fréquence de 1,2 GHz.

Les processeurs DSP modernes comme l'exemple précédent, sont de type *Very Long Instruction Word* (VLIW) [66, 67] et sont ainsi capable d'exploiter un parallélisme au niveau instruction. Un autre exemple comme le processeur *Trimedia* [68] de la société *NXP Semiconductors* propose aussi ce type de parallélisme. Tout comme dans ce dernier exemple, il n'est pas rare de trouver des architectures DSP contenant également des unités de type *Single Instruction Multiple Data* (SIMD) exploitant un parallélisme de données.

A l'instar des processeurs généralistes, nous assistons également à la démultiplication de coeurs de processeurs DSP sur une seule puce. En conservant l'exemple précédent du coeur *C66x*, nous pouvons citer le processeur TMS320c6670 [65] de la même société qui contient quatre coeurs DSP *C66x*.

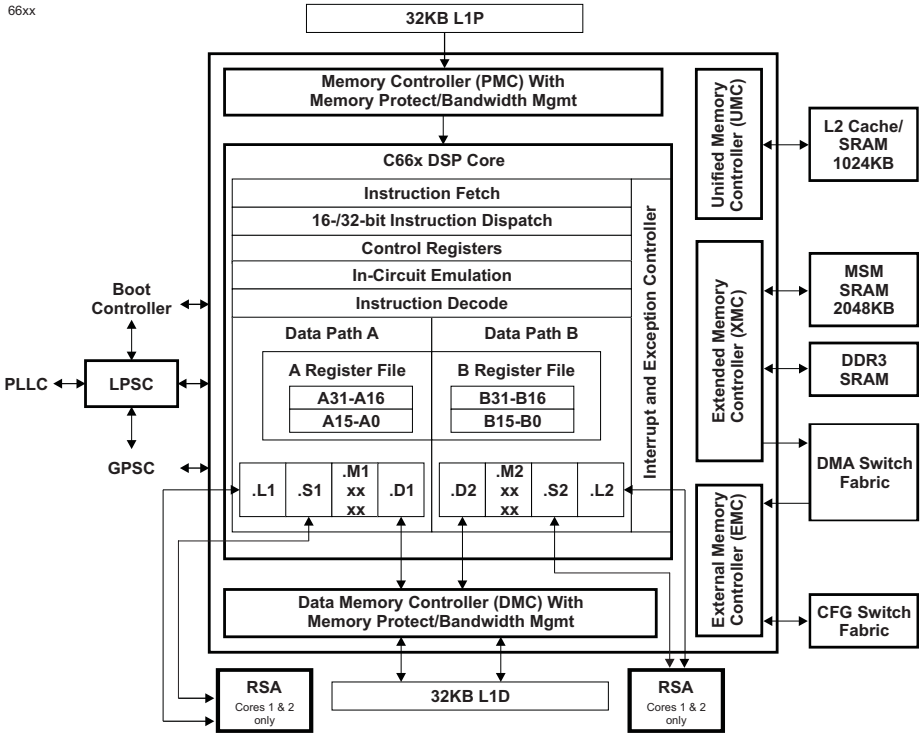


FIGURE 2.29: Architecture du coeur DSP *C66x* (*Texas Instruments*)

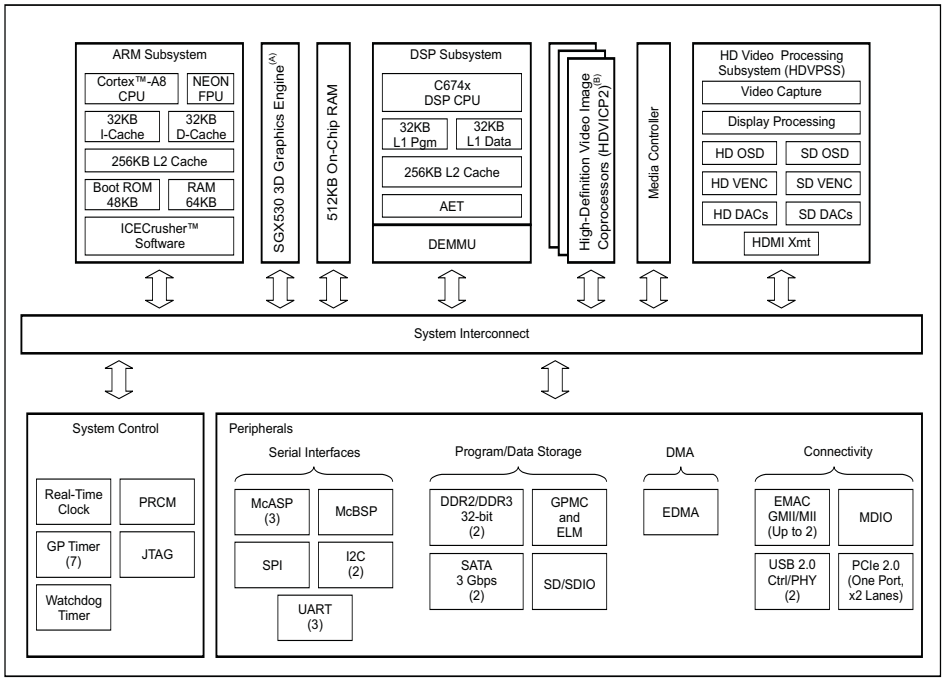


FIGURE 2.30: Architecture multi-coeur hétérogène *DaVinci* pour la vision

Dans le domaine de la vision embarquée portable, cette multiplication de coeurs reste peu efficace et coûteuse en consommation énergétique pour réaliser des fonctionnalités complexes comme la compression et la décompression vidéo, dans des standards nécessitant entre autres, des opérations d'estimation de mouvement. Un autre alternative consiste alors à intégrer dans un SoC des co-processeurs dédiés à certaines fonctionnalités dites critiques en performance. Les processeurs *DaVinci* [69] (2.30) de la société *Texas Instruments* proposent cette alternative en intégrant par exemple, sur une seule puce un coeur ARM *Cortex-A8* généraliste, un coeur DSP *C674x* et des co-processeurs dédiés pour réaliser différentes opérations dans le domaine de la vision (compression, acquisition, pré-traitement, affichage) sur des flux d'images de résolution 1080p à 60 Hz selon le fabricant.

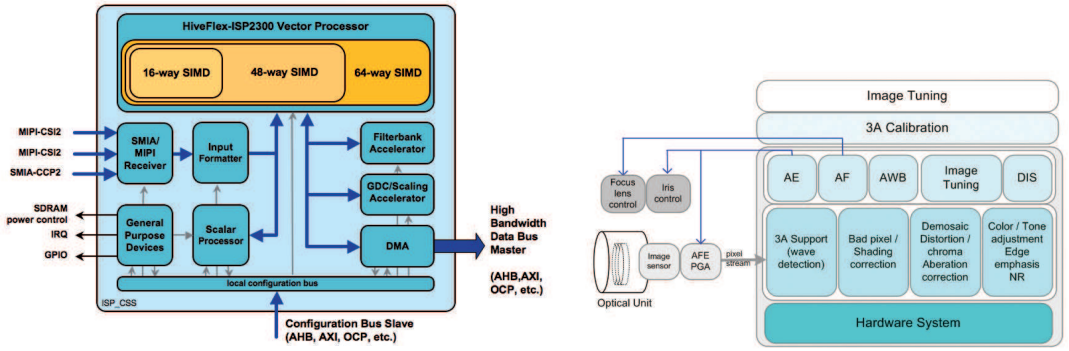
Nous avons vu dans les exemples précédents que les extensions SIMD sont fréquents pour réaliser des calculs dans le domaine du multimédia. De nombreux processeurs ont proposé d'exploiter ce type de parallélisme permettant d'appliquer sur plusieurs données différentes les mêmes instructions, par duplication d'unités de calcul au sein d'une même puce et sans pour autant dupliquer le contrôle et le programme mémoire. Nous avons vu précédemment, l'exemple des instructions *NEON* [70] pour les architectures ARM, mais nous pouvons encore citer *MMX/SSEx* [71] pour les processeurs *Intel* ou encore *Altivec* [72] pour les processeurs *PowerPC* de *IBM*. Ces instructions permettent ainsi de travailler sur des vecteurs de données.

Le principe de SIMD peut être établi à plusieurs niveaux : au niveau intra-processeur, en travaillant sur un jeu de données réduit, et au niveau inter-processeurs. Différentes architectures parallèles SIMD ont été proposées depuis de nombreuses années afin d'obtenir un gain en performance par rapport à des architectures séquentielles. Historiquement, les performances des machines SIMD, composées de plusieurs processeurs en parallèle, ont été rattrapés grâce à la montée rapide des technologies d'intégration permettant d'atteindre des fréquences de traitement de plus en plus élevées. La limite fréquentielle étant atteinte, l'étude des machines SIMD est redevenue d'actualité, appliquée au technologie de fabrication moderne, permettant à présent d'intégrer de nombreux coeurs de processeurs sur une seul puce.

Nous pouvons citer en exemple, l'architecture *VIRAM* [73] contenant un processeur MIPS 64 bits et un co-processeur vectoriel de 4×64 bits qui sont interconnectés avec des

bancs mémoires par un crossbar. Cette architecture est capable de délivrer, sur 32 bits, une puissance de calcul de 1,6 GFLOPS pour une consommation de 2 Watts.

Nous pouvons encore citer l'exemple des processeurs *HiveFlex* (figure 2.31) [74] de la société *Silicon Hive* qui sont dédiés aux équipements de vision portable en téléphone mobile et photographie numérique. Basé sur un coeur de processeur *HiveFlex 2300* de structure SIMD, le système *HiveGo* propose une solution capable de déployer une puissance de traitement supérieure à 270 Mpixels par seconde. Ce système est ainsi capable de traiter un flux HD 1080p avec une consommation sous le Watt. Notons cependant la nécessité d'utiliser de co-processeurs spécifiques pour supporter les contraintes de performance.



(a) HiveFlex 2300 (HW) (b) HiveGo CS3000

FIGURE 2.31: Architecture des processeurs SIMD de *Silicon Hive*

Pour des applications embarquées en vision automobile, le processeur *IMAPCAR2* [75] (figure 2.32), successeur de *IMAPCAR* de la société *Renesas*, propose une solution de 64 processeurs élementaires VLIW à 5 voies, capable de déployer 170 GOPS à une fréquence de 133 MHz. Il s'agit d'une architecture SIMD avec une organisation mémoire distribuée sur chaque PE. Les types d'applications implementés vont de la simple reconnaissance de marquage routier vers la reconnaissance de piétons sur la voie.

Toujours dans le domaine de la vision, le processeur *Xetal-II* [76] de la société *NXP* propose d'exploiter le parallélisme massif des données inhérent à l'acquisition des pixels sur une matrice CMOS. Comme expliqué dans la section précédente sur les capteurs, la technologie CMOS permet de combiner les process de fabrication du capteur et du processeur pour réaliser une solution intégrée. *Xetal-II* contient 320 processeurs élementaires de 16 bits et est capable de délivrer une puissance de 107 GOPS, à 84 MHz, pour une consommation de 0,6 Watts sur une surface de 74 mm^2 en technologie 90 nm. Cette méthode de parallélisme permet ainsi d'atteindre une bande passante de 1,3 Tbit/s.

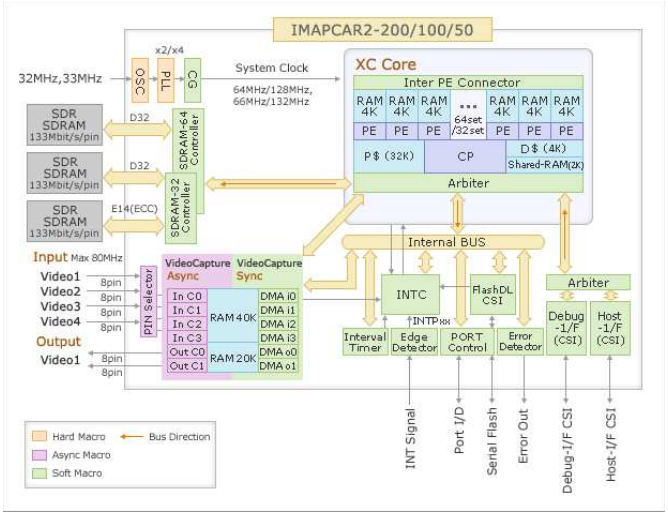


FIGURE 2.32: Architecture IMAPCAR2 (*Renesas*) [75]

Des recherches récentes se poursuivent encore afin d'abaisser sa consommation avec le Xetal-Pro [77], en modifiant la hiérarchie mémoire.

Les architectures multi-processeurs, dédiées à l'accélération graphique 3D, appelées *Graphic Processing Units* (GPUs), exploitent également le principe de structure massivement parallèle. L'architecture de ces processeurs ont été, à l'origine, uniquement dédiée au rendu 3D en temps réel, en combinant deux types de coeurs de traitement dédiés respectivement au calcul de pavage de l'espace par des triangles et à l'application des textures. Ces architectures introduisaient un déséquilibre fréquent de charge entre les deux types d'unités de calcul. C'est ainsi que la société *Nvidia* introduisit depuis fin 2006 une solution de processeur unifié, appelé *Unified Shader Architecture*, pour la série GeForce 8800, capables d'effectuer les deux types de traitement et ouvrant des portes vers d'autres types d'application par l'intermédiaire de l'API baptisé CUDA [78] (figure 2.33). Ces GPUs ont alors intéressé une communauté de recherche grandissante concernant la possibilité de détourner cette puissance de calcul pour effectuer des traitements plus génériques, appelés *General Purpose Processings on GPU* (GPGPU).

En exemple, le processeur GeForce 8800 GTX, issu de l'architecture G80 [79], est capable de délivrer une puissance de 518 GFLOPS pour une fréquence de 1,35 GHz. Ce type de GPU offre des performances attractives pour réaliser des applications de traitement d'image avec des opérations majoritairement pixéliques. Cependant, avec une consommation dépassant la centaine de Watts, ce type d'architecture n'est pas envisageable pour l'embarqué.

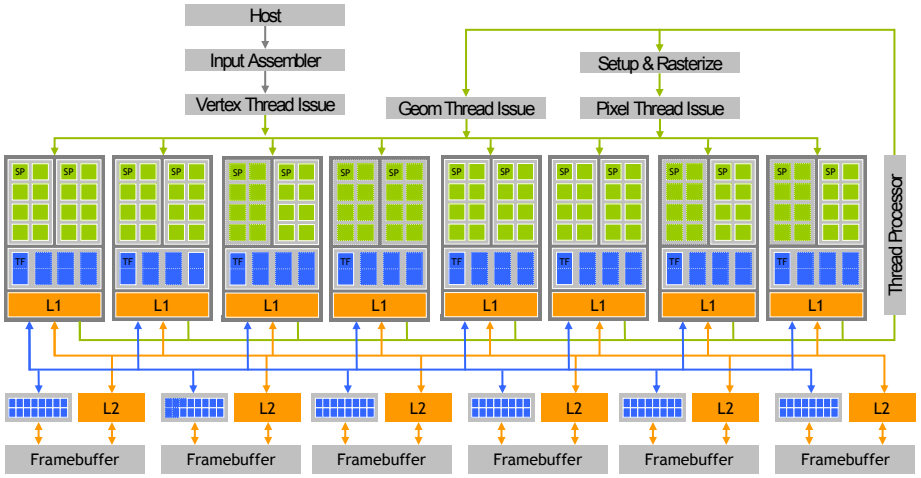


FIGURE 2.33: Architecture GPU G80 (*Nvidia*) [79]

L'architecture *Mali* de la société *ARM* est un exemple de GPU embarqué. La version *Mali 400 MP* [80] est capable de délivrer une puissance de calcul de 1,1 Gpix/s à une fréquence de 275 MHz, pour une surface inférieure à 5 mm^2. Les architectures GPU *PowerVR* de la société *Imagination Technologies* [81] sont les plus répandus et inclus dans la majorité des SoC pour des appareils portables, comme par exemple l'*OMAP* [82] de *TI* ou encore les processeurs *A4-5* de *Apple*. La série GX des PowerVR sont capables d'atteindre le Gigapixel/s pour une fréquence de 200 MHz. Ces GPUs embarqués sont cependant dédiés à de l'accélération 3D pour respecter les contraintes de consommation et de surface.

Dans le domaine de l'embarqué, les performances des GPUs ne sont pas aussi élevées que les modèles intégrés dans des cartes graphiques dédiées, afin de maîtriser la consommation énergétique. Cependant, à l'instar de l'évolution des processeurs généralistes, il ne serait pas surprenant de voir les architectures de GPU évoluer dans le domaine de l'embarqué grâce à la stimulation du marché multimédia portable et la demande croissante en accélération 3D, du simple affichage cartographique au domaine du jeux vidéo portable.

Les processeurs GPU peuvent également se classer dans une catégorie de processeurs dits *orientés flux*. En effet, avec un pipeline de traitement important, ils exploitent à la fois un parallélisme spatial et temporel. Ils sont donc naturellement adaptés pour le traitement de flux pixélique en limitant ainsi la quantité de mémorisation nécessaire.

D'autres exemples de processeurs *orientés flux* sont proposés par l'Université de *Standford*. Les processeurs *Imagine Stream Processors* ont initié l'architecture des processeurs *Storm* [83] de la société *Stream Processors*. L'architecture du processeur *Storm* est composée d'un processeur MIPS contenant la gestion globale avec un système d'exploitation Linux et un co-processeur contenant 16 processeurs VLIW d'ordre 5 permettant d'atteindre 512 GOPS à 800 MHz. Cependant, avec une consommation de l'ordre de la dizaine de Watts, ces processeurs ne sont pas adaptés à de l'embarqué portable.

2.4.2.3 Synthèse sur les architectures programmables

Nous avons vu dans cette section, différentes architectures programmables dont le défi permanent est d'offrir la maximum de flexibilité tout en approchant des performances temporelles requises, par différentes méthodes de parallélisation. Les solutions programmables, qu'elle soient de type généraliste ou spécialisée, font face à une limite fréquentielle technologique dans le domaine de l'embarqué.

Afin de monter en performance, les solutions actuelles reposent sur la multiplication des coeurs de traitement sur une seule puce de façon homogène mais aussi hétérogène. Ainsi, les SoCs modernes incluent généralement une combinaison hétérogène de type de processeur : processeur généraliste avec processeur DSP comme vu précédemment, processeur généraliste avec GPU comme le Tegra [84] de Nvidia. Ces différentes combinaison de solutions programmables imposent de nouveaux défis dans les stratégies d'implémentation d'une application afin de tirer profit du parallélisme à différents niveaux : tâches, instructions et données. Ces stratégies peuvent s'avérer complexes dans le cas de solutions hétérogènes.

Lorsque le domaine d'utilisation est bien ciblé comme celui de la vision, il est envisageable de restreindre le champ d'opérations afin d'optimiser certaines fonctions critiques avec des co-processeurs dédiés par exemple. Cependant, plus la granularité de ces co-processeurs est importante et plus le SoC devient spécialisé, le rendant ainsi rapidement obsolète dans un contexte d'évolution des applications comme exposé précédemment.

Il apparaît, de ce fait, un besoin permanent de modification de l'architecture pour atteindre des performances en temps et en surface. La technologie reconfigurable permet au concepteur de modifier physiquement l'architecture afin d'obtenir par exemple une

solution optimisée en chemin de données selon l'application. Elle offre ainsi la possibilité de personnaliser l'architecture de traitement au niveau des unités de calcul et des interconnexions.

2.4.3 Architectures reconfigurables pour l'embarqué

Par définition, une architecture *reconfigurable* est une architecture dont les ressources (calculs, interconnexions) peuvent être modifiées physiquement pour s'adapter à un traitement.

Un très grand nombre d'architectures reconfigurables ont été proposé depuis de nombreuses années [85] afin d'accroître la flexibilité et les performances de calcul.

Nous classons généralement les architectures reconfigurables selon plusieurs critères comme la granularité et le type de reconfiguration. On distingue aussi ces architectures en fonction du couplage avec un processeur embarqué et la forme des ressources de routage utilisés.

2.4.3.1 Granularité de reconfiguration

Nous distinguons dans la littérature deux niveaux de reconfiguration : le niveau *porte* et le niveau *fonctionnel*.

Le niveau *porte* correspond aux architectures reconfigurables dite à "grain fin" ("fine-grain") et le niveau *fonctionnel* correspond aux architectures reconfigurables dite à "grain épais" ("coarse grain") [10].

Une architecture reconfigurable à grain fin effectue des reconfigurations au niveau "bit" (niveau logique). La technologie *Field Programmable Gate Array* (FPGA) correspond à ce niveau de reconfiguration. Il est ainsi théoriquement possible de réaliser tout type de circuit numérique sur ce support. Un des points forts de ce type de technologie est donc sa haute flexibilité. Cependant, ses performances sont moyennes en ce qui concerne son temps de reconfiguration car il faut en effet reconfigurer tous les bits logiques de ce type de support. Il en résulte un flux de reconfiguration (*bitstream*) de taille importante. Afin de minimiser ce temps, une solution serait de reconfigurer partiellement ce support ou

d'élever technologiquement la granularité de reconfiguration. C'est l'objectif des solutions reconfigurables à grains épais.

Les ressources de calcul d'une architecture à grain épais, ne travaillent plus au niveau "bit" mais au niveau "mot" (groupe de bits). Ces unités de calcul sont généralement appelées *Coarse Grain Reconfigurable Units* (CGRUs) [10]. Les modifications du chemin de données ciblent alors les interconnexions, sous forme de bus, par reconfiguration ou programmation des connexions entre les unités de calcul afin d'optimiser les échanges de données. Il en résulte un temps de reconfiguration minimisé et un espace de stockage en mémoire réduit du plan de configuration appelée *contexte*. Il est à noter également que la consommation de l' opération de reconfiguration s'en trouve également réduite. Nous pouvons remarquer qu'il est de façon évidente impossible de réaliser un architecture à grain épais universelle pour tout type de calcul. Cette solution ne peut être efficace qu'en délimitant le domaine d'utilisation avec un certain nombre d'opérateurs fréquemment utilisés dans ce domaine, appelés *kernels* comme une FFT ou une DCT en traitement du signal. Le travail sera de dimensionner les unités de calcul (type, taille, nombre) mais aussi le système d'interconnexion.

2.4.3.2 Type de reconfiguration

Nous distinguons deux types de reconfiguration : la reconfiguration statique et la reconfiguration dynamique.

Dans le cas d'une reconfiguration statique, les données de reconfiguration et les données à traiter sont distribuées via des chemins différents et à des instants disjoints. La reconfiguration implique l'arrêt du composant reconfigurable afin de pouvoir recharger le contexte puis relancer les nouveaux traitements. Dans le cas d'un système de vision, cette méthode implique un arrêt du composant reconfigurable avec une coupure du flux pixélique si celui-ci ne transite que par cet unique composant.

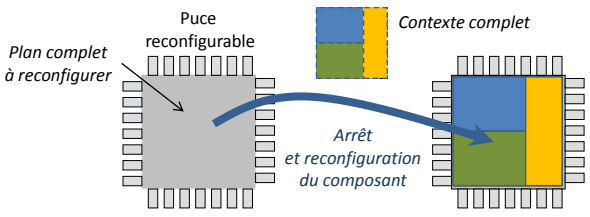


FIGURE 2.34: Méthode de reconfiguration statique

Dans le cas d'une reconfiguration dynamique, les données et les configurations sont distribués simultanément. Les phases de reconfigurations et exécutions sont ainsi concurrentes. Cette méthode n'implique pas un arrêt du composant reconfigurable qui est capable de modifier totalement ou partiellement sa surface de calcul. Nous qualifions une architecture capable de se reconfigurer de façon autonome d'auto-reconfigurable ("self-reconfigurable") [86, 87]

Dans le cas d'une méthode partielle, on conserve une partie de région de la surface reconfigurable en cours de fonctionnement et on reçoit des données de reconfiguration sur une autre région [88].

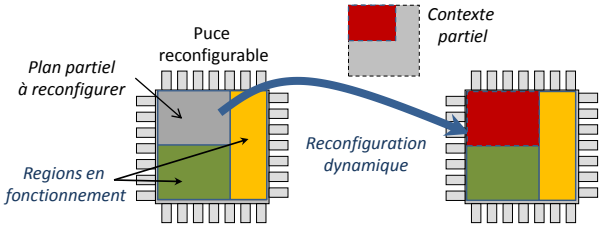


FIGURE 2.35: Méthode de reconfiguration dynamique partielle

Dans le cas d'une méthode totale, on reconfigure entièrement la région pendant l'exécution d'une même application et ceci afin de changer le traitement en cours. Ce type de reconfiguration nécessite des ressources mémoires et logiques particulières capable de stocker plusieurs contextes sur le même plan reconfigurable afin de pouvoir basculer rapidement de configuration. Dans la littérature, cette méthode est aussi appelée reconfiguration « multi-contexte » [89]. Cette solution nécessite un temps de reconfiguration très court selon le type d'application et impossible à masquer en temps contrairement à la méthode partielle. Nous retrouvons un principe similaire dans l'architecture flexiFLASH du composant XP2 [90] proposé par la société Lattice qui combine un plan mémoire à technologie Flash avec un plan mémoire SRAM de configuration sur une même puce, permettant de basculer de contexte en 1 ms (figure 2.36).

2.4.3.3 Couplage avec un processeur

Une architecture reconfigurable implique généralement la nécessité d'un processeur afin de contrôler les différentes configurations, le partitionnement et le séquencement des opérations. On peut ainsi caractériser les architectures reconfigurables suivant le couplage avec le processeur embarqué. Le couplage le plus fort est celui considérant une surface

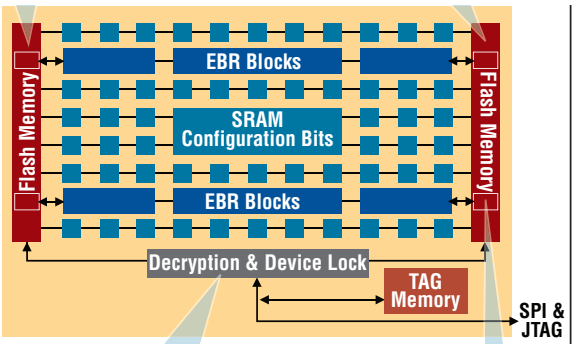


FIGURE 2.36: Architecture flexiFLASH du FPGA XP2 (Lattice) [90]

reconfigurable pour les *unités fonctionnelles* du processeur. De façon moins intrusive, le composant reconfigurable peut être placé comme un *co-processeur* correspondant à un circuit spécialisé qui est sollicité pour des tâches spécifiques. Nous pouvons citer un exemple proposé par Claus dans [91] dans la conception d'un co-processeur reconfigurable en traitement d'image dans un contexte d'assistance automobile. La solution la plus classique étant celle considérant un composant reconfigurable comme un *périphérique externe* interconnecté avec un processeur par un bus externe. Nous pouvons citer en exemple le processeur embarqué *Stellarton* [92] combinant sur une seule puce un processeur généraliste *Intel Atom* [61] et une FPGA *Altera* (Arria II GX). Cette dernière solution introduit une latence importante dans la communication entre processeur et logique reconfigurable.

Un dernier couplage étant celui considérant un *processeur embarqué dans la logique reconfigurable* ce qui permet de simplifier les communications. Cette solution a été de plus en plus répandue avec la maturité des technologies reconfigurables à grain fin de type FPGA, capable de réaliser des SoC complet appelés plus précisément *System-on-Programmable Chip* (SOPC). Il est à noter que les FPGAs atteignent des densités telles qu'ils sont capable de synthétiser directement un système programmable de type processeur voire multi-processeurs. Ces processeurs réalisés à partir de logique reconfigurable ne permettent néanmoins pas actuellement d'atteindre des fréquences élevées et restent de l'ordre de la centaine de MHz. Toutefois, certains composants reconfigurables peuvent accueillir un processeur physique capable de traiter à plus haute fréquence les opérations. Nous pouvons citer en exemple le ZYNQ-7000 EPP [93] (figure 2.37) proposé par la société Xilinx. Ce composant contient un processeur ARM-Cortex A9 double coeur pouvant atteindre une fréquence de 800 MHz.

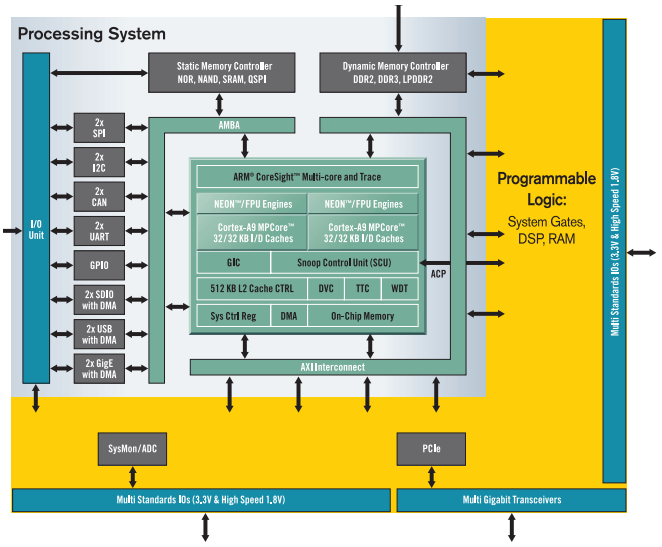


FIGURE 2.37: Architecture du ZYNQ-7000 EPP (Xilinx) [93]

2.4.3.4 Exemples de solution reconfigurable pour la vision

De nombreuses solutions d'architecture reconfigurable ont été proposées depuis plusieurs années [85]. La majorité de ces architectures sont capables d'effectuer des calculs de tout type selon la granularité de reconfiguration. Elles peuvent ainsi aussi s'appliquer pour des applications de vision et différents portages ont été étudiés.

Comme montré dans la section 2.3 présentant les applications, nous constatons que l'ordre d'enchaînement des opérations à effectuer sont bien définis par domaine : restauration, analyse, amélioration, restitution. Partant de ce principe, il est envisageable de définir une structure reconfigurable respectant un enchaînement classique des opérateurs de traitement d'image pouvant être reconfigurables. Il en résulte une architecture reconfigurable, dite gros-grain, composée d'une unité programmable se chargeant de la gestion globale et des unités reconfigurables au niveau des unités de calcul et/ou au niveau des interconnexions.

C'est l'approche proposé, en exemple, par le processeur *CRISP* [94] de l'université de Taiwan *NTU* (figure 2.38).

Son architecture est constituée d'un processeur généraliste de contrôle couplé avec un système d'interconnexion et des étages de calcul reconfigurables. Nous pouvons voir que cette architecture exploite également un parallélisme temporel des tâches avec un assemblage possible des unités de calcul sous forme d'un pipeline grâce au système d'interconnexion reconfigurable. Cette architecture orienté flux, offre des performances de

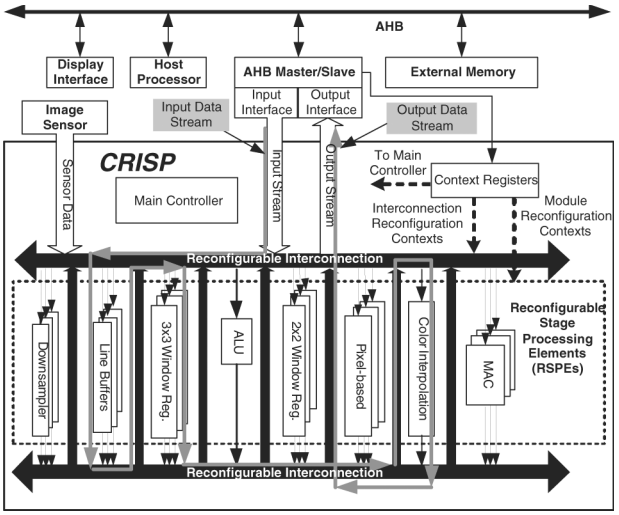


FIGURE 2.38: Architecture CRISP [94]

calcul de 10 images/seconde pour des résolutions atteignant 11 Mpixels, et 55 images/-seconde pour une résolution haute définition 1080p (1920 × 1080) pour seulement une fréquence de 115 MHz.

Cette architecture a été suivie par une seconde version *CRISP-DS* (*Dual Stream*) [95] capable de traiter deux flux en parallèle. Ses performances ont été démontrées récemment dans le cadre d'une application de traitement d'image mettant en oeuvre un opérateur HDR fusionnant deux images [18].

Ce type d'architecture présente cependant quelques inconvénients car, étant optimisée pour une chaîne de traitement, elle ne pourrait pas être efficace pour d'autres opérations comme la compression ou l'analyse d'image. Ce processeur n'est aussi utilisé en pratique que sous forme d'un accélérateur d'un processeur généraliste. Se trouve également posé le problème de la configuration des différents étages de calcul qui, selon les auteurs, restent encore manuelle et nécessite un environnement logiciel associé.

Nous pouvons également citer un autre exemple d'architecture reconfigurable gros-grain comme *ADRES* [96] (figure 2.39), qui est composée d'une partie VLIW programmable et d'un partie reconfigurable composée d'une matrice de processeurs élémentaires dont le système d'interconnexion est reconfigurable pour différentes topologies. Cette architecture propose une puissance de 40 MOPS/mw.

Dans le domaine des architectures reconfigurables à grain fin, la densité d'intégration

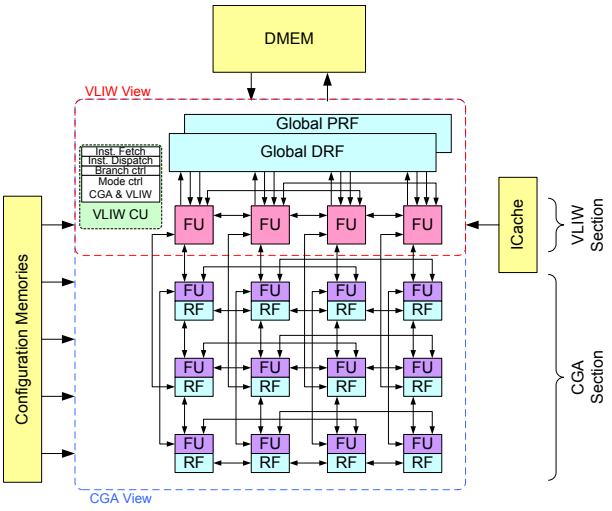


FIGURE 2.39: Architecture ADRES [97]

croissante des FPGAs a permis de faire émerger de nombreuses propositions d'architectures reconfigurables pour la vision embarquée [98]. Nous pouvons citer par exemple l'architecture ARDOISE (*Architecture Reconfigurable Dynamiquement Orientée Image et Signal Embarqué*) [99] qui est une solution à la fois multi-cartes et multi-FPGAs (ATMEL AT40K) reconfigurable dynamiquement. Pour des systèmes mono-FPGA, Sedcole [100] a proposé, par exemple, d'implémenter sur une seule puce une version de l'architecture *Sonic* dédiée au traitement d'image. Il s'agit d'une architecture multi-processeurs élementaires (figure 2.40) pouvant être interchangés par reconfiguration dynamique.

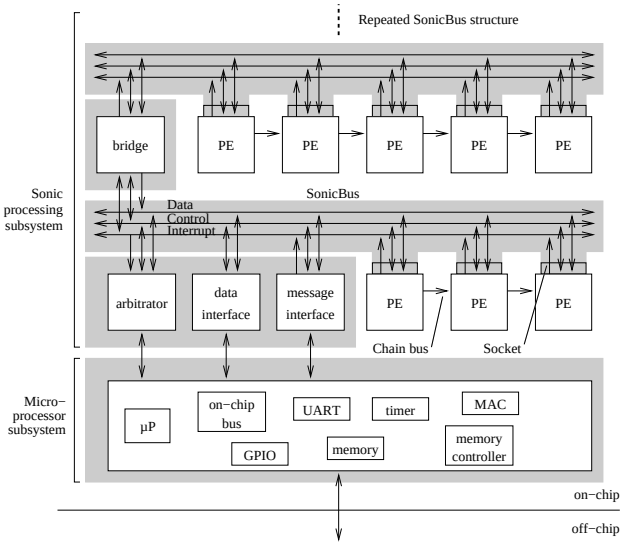


FIGURE 2.40: Architecture *Sonic-on-a-Chip*

Nous pouvons remarquer que le système d'interconnexion prévoit une communication point-à-point (*chain bus*) des unités de calcul (PEs) de façon pipelinée afin de minimiser

les accès mémoires et de traiter sur le flux.

Cette architecture exploite le principe de conception de systèmes dérivés (”derivative systems) [100] en se basant sur une plate-forme commune pouvant être personnalisée selon les contraintes de surface du système de vision. Cette approche est la plus utilisée dans les SOPC modernes implémentés sur des cibles reconfigurables à grain fin comme les FPGAs. L'idée principale consiste à se fixer un réseau d'interconnexion et à ajouter des unités de calcul (PE). Selon la méthode de changement de fonctionnalité du PE les modifications peuvent se faire avant synthèse ou dynamiquement par reconfiguration.

Il est tout à fait possible de synthétiser une solution programmable de type processeur dans une logique reconfigurable à grain fin comme le FPGA. Les processeurs *Nios* ou *MicroBlaze* en sont des exemples de processeurs généralistes synthétisables. Leur fréquence de fonctionnement reste faible de l'ordre de la centaine de MHz en raison des limites technologiques. Leur rôle principale dans un SOPC est ainsi réservé principalement au contrôle. Il est cependant tout à fait faisable de définir des fonctions accélératrices pour ces processeurs. Nous pouvons citer l'exemple de l'architecture Terapix [101] de la société *Thalès* qui est un processeur SIMD synthétisé complètement sur un Virtex-4 SX55 proposant une puissance de calcul de 40 GOPS.

2.4.3.5 Synthèse sur les architectures reconfigurables

Les architectures reconfigurables imposent, de par leur technologie de fabrication, un surcoût en surface par rapport à des solutions câblées. Cependant, elles offrent au concepteur des possibilités de personnalisation de l'architecture et ses degrés de liberté sont fixés par la granularité et le positionnement de la reconfigurabilité (unité de calcul, interconnexion). Nous pouvons classer ces architectures suivant la granularité de reconfiguration, pouvant être à grain fin ou épais ; le type de reconfiguration, pouvant être statique ou dynamique ; et le couplage avec un processeur généraliste, décrivant la dépendance et le positionnement des parties reconfigurables.

Un grand nombre de solutions de vision embarquée reconfigurables exploitent le principe de conception plate-forme (“platform-based design”) pour réaliser des solutions dites dérivées (*derivative systems*) [100]. Ce principe est proposé par la majorité des architectures de type SoPC à base de technologie FPGA dans un but de capitalisation et réutilisation

des développements sur les unités de calcul et sur le système d'interconnexion. Il reste le meilleur compromis pour assurer la pérennité du système et être capable de traiter plusieurs types de profil selon les contraintes d'encombrement ou de consommation avec des systèmes dérivés. Chaque système conçu sur une même base d'architecture peut être mappé par exemple, sur des ressources à base de technologies reconfigurables en constante croissance en terme de densité et de fréquence. En particulier, les composants FPGA atteignent des densités de plusieurs milliards de transistors tel le composant *Stratix V* [102] de la société *Altera* atteignant 3,9 milliards de transistors, et évolueront vers des fréquences de fonctionnement plus importantes. Certaines cibles sont même annoncées pour des fréquence de fonctionnement de l'ordre du GHz comme le *Speedster22i-HP* [103] de la société Achronix utilisant une technologie de gravure 22 nm de la société *Intel*, atteignant une fréquence de 1,5 GHz.

La densité des surfaces FPGA étant croissante et les fréquences de fonctionnement stagnantes, nous observons toutefois que la tendance actuelle, dans l'évolution de ces composants, est de monter en granularité. Il est maintenant courant de trouver bien plus que des cellules reconfigurables dans une seule puce FPGA : mémoire embarquée, transceivers, unités de calcul de type *Multiplieur-ACcumulateur* (MAC), contrôleur mémoire et même un processeur généraliste physique comme montré dans les exemples précédents. Ces types de puce constituent une solution séduisante pour réaliser des SoC à faible coût en considérant la production d'un équipement de vision en quantité réduite, ce qui est le cas de la majorité des systèmes de vision dans le domaine de la défense.

2.5 Conclusion

La conception de l'électronique d'un système de vision embarqué moderne est un problème complexe. Elle nécessite de tenir compte du choix des capteurs d'image, de la disparité des types d'application et des contraintes de performance en temps et en surface.

D'un point de vue développement et pérénnité, la solution consistant uniquement à décrire des architectures câblées à communication point-à-point complètement dédiées, devient peu efficace dans le contexte actuel où le marché de la vision embarquée portable

est en constante évolution, avec une variété de capteurs et une variété d'applications de complexité croissante.

A l'opposée, le fait de n'utiliser uniquement que des solutions purement programmables ne constituent toujours que des solutions à court terme. En outre, plus ce système programmable est généraliste dans ses unités de calculs et plus son efficacité est diminué pour un domaine de traitement particulier. L'efficacité et la pertinence d'une solution ne sont véritablement atteints qu'avec un cadrage du domaine d'application avec une famille de traitement. Dans le domaine des solutions programmables, nous assistons à une tendance à multiplier les coeurs sur une seule puce, qui peuvent être hétérogènes [104, 105]. Cette tendance impose de nouveaux défis au niveau des stratégies de parallélisation des calculs et de la gestion des accès aux données.

Par ailleurs, la réalisation de son propre processeur d'image spécialisé permet d'atteindre des performances ciblées mais implique un lourd investissement de développement à la fois logiciel et matériel pour aboutir à une solution rapidement obsolète si la scalabilité de sa structure est faible par rapport à l'évolution croissante des technologies silicium.

Une architecture de type SOPC à base de technologie FPGA est la solution la plus appropriée pour un contexte nécessitant de faire évoluer en permanence des applications de traitement d'image, avec le développement d'unités de calcul de complexité croissante. En outre, une plate-forme SOPC autorise différentes déclinaisons d'architectures de calcul, ce qui permet d'adresser différentes familles de système de vision qui sont définies selon des contraintes de performance, d'encombrement et de consommation. Bien qu'engendrant un surcoût naturel en terme de surface silicium, un composant de type FPGA permet de réaliser des systèmes de vision de faible coût. Il s'agit d'une cible particulièrement adaptée dans le domaine de la défense dans un contexte où les volumes de production sont faibles pour un système de vision embarqué.

Néanmoins, bien que la puissance de calcul des architectures a été démontré par de nombreuses propositions [9, 10], le verrou majeur se situe principalement au niveau du *système d'interconnexion* qui n'est pas suffisamment flexible, en particulier pour le transfert des flux de pixels et l'accès à la donnée.

La multiplicité de capteurs [16] apporte une difficulté supplémentaire à la conception du système d'interconnexion. La majorité des systèmes multi-capteurs, ciblant généralement

des applications de stéréovision, de fusion ou de réhaussement HDR, utilisent des solutions de communication dédiées et ainsi peu flexibles à des modifications d'applications [17, 18].

De l'étude des applications dans cette section, nous avons constaté un recouvrement des types d'opérations utilisés pour des applications de visualisation usuelles à chaque type de capteur. Les différences majeures résident dans le séquencement de ces opérations et de la présence d'opérations spécifiques au capteur.

En considérant qu'un concepteur dispose d'un ensemble d'unités de calcul optimisées (IPs) pour réaliser l'ensemble des opérations nécessaire aux applications, il convient de travailler sur des solutions d'adaptation du système d'interconnexion. La problématique repose sur le fait de déterminer un système d'interconnexion capable d'être paramétré pour acheminer plusieurs types de flux en parallèle entre les unités de calcul, qui sont principalement orientées flot de données d'après notre étude.

Nous nous intéressons ainsi dans le chapitre suivant aux différents systèmes d'interconnexion utilisés dans les SoC. Nous discuterons également de leur utilisation pour un système de vision embarquée.

Chapitre 3

Systèmes d'interconnexion dans un SoC

3.1 Introduction

Les performances d'une architecture implémentée dans un SoC dépendent fortement du système d'interconnexion et du protocole de communication entre les unités de calcul. Avec la technologie d'intégration croissante, la conception d'un système d'interconnexion efficace est critique pour exploiter pleinement le nombre et la puissance de traitement des unités de calcul dans un même circuit.

En particulier, l'augmentation permanente des densités de système à base de logique reconfigurable à grain fin de type FPGA, offre de très grandes possibilité de parallélisation et d'intégration d'unités de calcul, pour réaliser des solutions d'interconnexion complexes entre les éléments.

Les systèmes d'interconnexion sur puce sont généralement des adaptations architecturales à échelle réduite de solutions existantes à plus grande échelle, comme par exemple des clusters de processeurs sur des cartes électroniques communiquant sur un bus partagé ou encore un réseau de processeurs sur une même carte.

Dans ce chapitre, nous effectuons une description succinte de différents systèmes d'interconnexion utilisés dans un SoC. Nous évoquons en particulier l'émergence des systèmes d'interconnexion de type *réseau sur puce* (NoC) depuis une dizaine d'années. Enfin, nous

discutons de l'adéquation de ces solutions d'interconnexion par rapport au contexte de vision embarquée et à notre problématique exposée au chapitre précédent.

3.2 Variété des systèmes d'interconnexion

Il existe différents types de systèmes d'interconnexion utilisables dans un SoC dont le *point-à-point*, le *bus partagé* ou *hiérarchique*, le *crossbar* et le *réseau* sur puce [106].

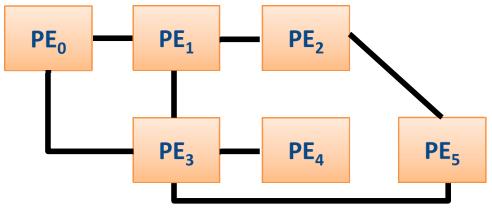
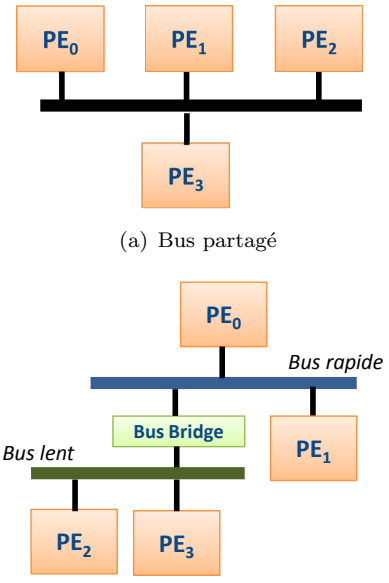


FIGURE 3.1: Systèmes d'interconnexion point-à-point

L'approche *point-à-point* (figure 3.1) est la solution la plus directe et la plus simple. Il s'agit de celle qui est utilisée pour concevoir les architectures câblées évoqués au chapitre précédent. Elle consiste à connecter les différents unités de calcul d'un système avec des fils dédiés et exclusifs pour les échanges de données. Par conséquent, cette approche est efficace pour des systèmes à bande passante élevée ce qui est le cas dans le domaine de la vision. Elle offre une très grande possibilité de parallélisation mais implique une faible réutilisation des unités de calcul du fait de la rigidité des connexions. Il en résulte une très faible flexibilité de communication entre les unités de calcul. Cette solution reste cependant convenable pour des systèmes à faible nombre d'unités. Ainsi, faire évoluer un système avec cette approche, nécessite de complexifier les connexions, en rajoutant de façon croissante des liaisons entre les unités. Cette méthode devient, de manière évidente, difficilement gérable avec une démarche d'intégration croissante des unités de calcul pour des raisons de dépendances physiques entre les liaisons et de synchronisation entre les signaux.

L'approche par bus *partagé* (figure 3.2(a)) est une technique très utilisée pour interconnecter des unités de calcul. Cette approche convient également pour des systèmes à faible nombre d'unités de calcul. Par exemple, nous pouvons citer le bus standard ARM *AMBA* [107] qui est très répandu dans les SoCs. Différents bus sont généralement connectés sous forme *hiérarchique* (figure 3.2(b)) en regroupant des unités selon les contraintes



(a) Bus partagé

(b) Bus Hiérarchique

FIGURE 3.2: Système d'interconnexion de type Bus

en bande passante. Un bus est généralement composé de lignes de données, de lignes de contrôle et d'un arbitre. Ce type d'approche présente l'avantage d'avoir une mise en oeuvre simple à faible coût mais est cependant très limité en terme de performance car ce type de communication ne permet de faire communiquer qu'*un seul module à la fois* selon l'arbitrage. Il en résulte des formations de goulot d'étranglement de donnée très fréquents avec l'augmentation des unités connectés.

Un approche coûteuse consiste à utiliser un *crossbar* pour interconnecter les unités de calcul. Le principe consiste à définir une matrice de multiplexeurs permettant à toute unité du système de communiquer avec une autre, de façon la plus performante possible de manière *point à point*, et en autorisant ainsi des communications en parallèle. Cette approche est très coûteuse en surface mais convient pour des systèmes avec un nombre d'unités de calcul réduit. Un compromis consiste également à réaliser partiellement cette matrice en fonction des besoins de communication si ils sont prévisibles. En exemple, nous pouvons citer l'architecture du processeur reconfigurable, appelé *Reconfigurable Operators for Multimedia Applications* (ROMA) [108], proposé récemment par le CEA-LIST. Cette architecture utilise un crossbar comme système d'interconnexion entre des unités de calcul à grains épais et différents bancs mémoires.

Pour un SoC dédié à la vision, tous les types d'interconnexion présentés dans cette section sont utiles selon le profil et la finalité du système. Généralement, différents types

d'interconnexion sont combinés dans un même SoC, en utilisant les avantages de chacun. Les communications *point-à-point* sont par exemple réservées pour les parties critiques en terme de temps et les communications par *bus partagé* sont plutôt dédiées pour des périphériques lents et peu demandeurs en bande passante. L'utilisation de *crossbar incomplet* permet de résoudre partiellement les exigences en performances pour des SoCs actuels dédiés à la vision. Cependant, ces solutions se révèlent très rapidement limités en nombre d'unités de calcul.

Avec des technologies d'intégration croissantes, il est a présent envisageable d'implémenter dans un SoC des systèmes d'interconnexion plus avancés comme un *réseau* sur une seule puce.

3.3 Réseaux sur puce

3.3.1 Définition

Depuis une dizaine d'années [12, 13], les premiers concepts et prototypes de réseau de communication sur puce, appelé *Network-on-Chip* (NoC), sont apparus.

Selon la définition proposée par Dally [14], un réseau d'interconnexion correspond à un système permettant de transporter des données entre des terminaux. La figure 3.3 illustre un exemple avec trois terminaux T0, T1 et T2.

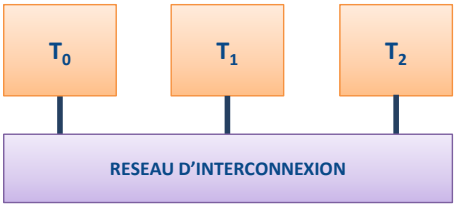


FIGURE 3.3: Réseau d'interconnexion du point de vue fonctionnel avec trois terminaux

Les terminaux se communiquent en se transmettant des *messages* (données) au travers du réseau. Dans notre cas, un terminal correpond à un point *source* ou un point de chute (*sink*) des données. Le réseau peut effectuer plusieurs connexions simultanées entre les terminaux autorisant ainsi plusieurs communications en parallèle et les modifier à tout instant. Un réseau est défini comme un *système* car il est composé de différentes ressources : mémoire, canal de communication et d'un élément de commutation de données appelé *routeur* qui s'organisent pour transmettre les messages entre les terminaux. Nous

définissons un *canal* comme un ensemble de fils interconnectant les ports d'entrée et de sortie des routeurs permettant de transporter les données.

A l'échelle du SoC, considérons à présent que chaque terminal correspond à une unité de calcul. L'approche par *réseau* d'interconnexion, illustrée par la figure 3.4, consiste à paquetiser les messages pour les transporter d'une unité source vers une unité destination, à travers un réseau de routeurs reliés par des canaux physiques de communication.

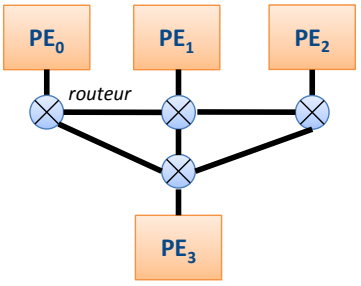


FIGURE 3.4: Système d'interconnexion de type réseau

3.3.2 Structure d'un message

Chaque *message*, illustré par la figure 3.5, est fractionné en plusieurs *paquets* de données [14]. Le paquet est l'unité d'information du réseau de communication. Ils sont en général composés d'un en-tête (*header*) contenant des informations de contrôle et de la donnée utile à transporter. Afin de pouvoir être transmis dans le réseau, ces paquets sont divisés en paquets élémentaires appelés *flits* (contraction de *flow control digits*). Ces *flits* peuvent être encore subdivisés en *phits* (contraction de *physical units*) correspondant à une unité de donnée qui peut être transférée physiquement à travers un canal de données entre deux routeurs en un cycle d'horloge. Les performances (i.e. latence, bande passante, consommation) d'un réseau sur puce dépendent dans un premier temps du choix de la granularité de chaque unité de données (message, flits, phits).

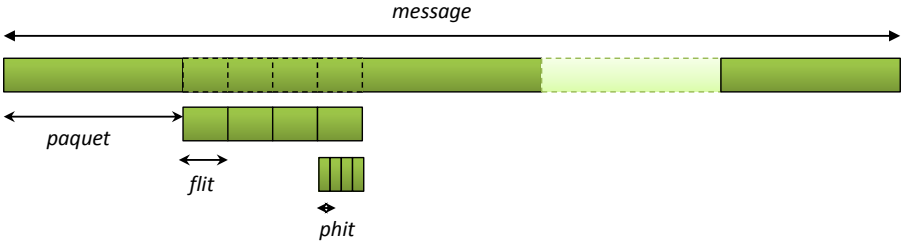


FIGURE 3.5: Structure d'un message dans un réseau NoC [14]

De nombreuses études ont été effectuées depuis une dizaine d'années, pour porter et évaluer différentes solutions d'interconnexion de systèmes multi-processeurs vers cette nouvelle approche de réseau de communication. C'est le cas par exemple pour l'architecture *Chameleon* dont les communications ont été évaluées dans un contexte de réseau de processeurs sur puce [15].

La spécification d'un réseau se base sur trois points essentiels : la *topologie* du réseau qui précise la structure du réseau, la *technique d'aiguillage* qui spécifie les mécanismes d'accès aux ressources et l'*algorithme de routage* qui détermine la gestion du trafic (distributivité des données) dans le réseau. Ces points sont inter-dépendants. Par exemple, le choix d'une topologie implique un choix d'algorithmes de routage propres à la topologie. Ainsi, toute la difficulté du concepteur consiste à réaliser des compromis en terme de consommation, performance, robustesse et complexité.

3.3.3 Topologie du réseau

La terminologie du réseau de communication reprend des termes définis dans le domaine de la *théorie des graphes*.

La topologie du réseau correspond à l'organisation de l'interconnexion des unités par des canaux de communication. Elle est définie comme un ensemble N de noeuds (*vertices*) connectés par un ensemble C de canaux (*edges*). Un canal $c_{x,y}$ tel que

$$c_{x,y} = (x, y) \in C \mid x, y \in N \tag{3.1}$$

permet de connecter un noeud x à un noeud y. La topologie d'un réseau est ainsi représenté par le graphe G.

$$G = (N, C) \tag{3.2}$$

Il n'existe pas de topologie idéale pour tout système et la difficulté du concepteur consiste à réaliser un ensemble de compromis entre la connectivité, le coût en ressources et aussi les choix de la granularité des messages dans le réseau.

Au niveau architecturale, nous appelons *noeud* le couple unité de calcul (PE) avec son routeur de donnée associé, comme illustré en figure 3.6.

Le *degré de liberté* d'un noeud correspond aux nombres de canaux permettant à un noeud de communiquer avec des noeuds voisins. Il correspond à la somme du nombre de canaux de communication entrants avec le nombre de canaux de communication sortants. Ce paramètre affecte de façon importante la complexité du réseau et la taille du routeur de communication.

La distance entre les noeuds est mésurée généralement en nombre de sauts (*hops*). Plus généralement, il peut y avoir plusieurs chemin de données possible entre deux noeuds x et y. Nous appelons $p_{x,y}$ le chemin de donnée (*path*) entre x et y correspondant à un ensemble ordonné de canaux. Le nombre de canaux contenu dans cet ensemble correspond alors à la longueur du chemin.

$$p_{x,y} = \{c_0, c_1, ..., c_k\} \mid \forall i \in \mathbb{N}, c_i \in C \tag{3.3}$$

Selon leur topologie, les réseaux de communication sont classés suivants trois catégories : les réseaux *directs*, *indirects* et *irréguliers* [109].

Dans le cas d'un réseau *direct*, chaque noeud est connecté aux noeuds voisins par des liaisons point-à-point. Les réseaux directs les plus connus sont les réseaux *mesh* (grille ou maillé), *torus*, *folded torus* et *octagon*.

La topologie de type *mesh* est la plus courante pour des systèmes multi-PEs [110] et pour les réseaux sur puce. Un exemple de maille 3x3 est présenté en figure 3.6.

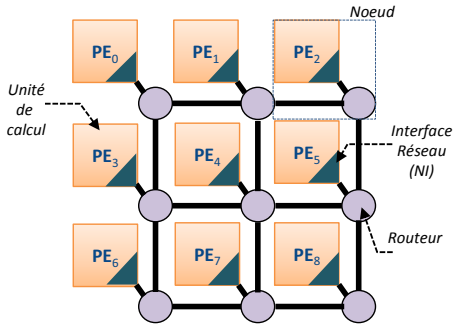


FIGURE 3.6: Exemple de réseau NoC 3x3 en topologie grille (mesh)

Chaque *noeud* correspond à une unité de calcul, une interface réseau (*Network Interface*(NI)) et un routeur. L'interface réseau permet de découpler l'unité de calcul du

réseau de communication. Il s'occupe de mettre sous forme de paquets les données issues de l'unité afin de le transmettre sur le réseau. Le routeur implémente la stratégie de routage nécessaire pour transmettre les données d'une unité de calcul à une autre. Généralement, un routeur est composé de tampons mémoires (*buffers*) en entrée et en sortie permettant de gérer plusieurs paquets de données en parallèle, d'un crossbar pour aiguiller un paquet entrants vers une des sorties du routeur et d'un contrôleur permettant d'arbitrer les différentes requêtes d'aiguillage selon la donnée en entrée et l'état des routeurs voisins. Ce contrôle global est régit par un algorithme de routage.

Dans un réseau de type *indirect*, les routeurs de chaque PE ne sont pas reliés directement mais au travers d'un ou plusieurs routeurs qui sont réliés par des liaisons point-à-point. Dans les réseaux sur puce, les topologies de type *tree* (arbre) ou *butterfly* (papillon) sont les plus utilisées. Dans une topologie de type *tree*, les noeuds de calcul correspondent aux noeuds feuilles de l'arbre (extrémités du réseau de communication). Il est évident que cette topologie pose des incovénients en terme de bande passante et de goulot d'étranglement de données situé principalement au noeud *root* (racine). Pour pallier à ce problème, une amélioration consiste à utiliser une topologie de type *fat tree* qui conserve la même structure que le *tree* mais dont les canaux de communication augmente à mesure où l'on se rapproche du noeuds racine. L'objectif étant d'augmenter la bande passante progressivement vers le noeud racine diminuant ainsi les goulots d'étranglement de données.

Enfin, les réseaux *irréguliers* sont en général un mélange entre réseaux directs et indirects. Ils sont utilisés pour des applications très spécifiques. Ils sont cependant beaucoup plus complexe en terme de prédiction de performance en temps du fait de la topologie irrégulière.

Quelle que soit la topologie choisie, un réseau sur puce est dit *homogène* ou *hétérogène*. Un réseau est dit *homogène* lorsque tous les unités de calcul du réseau sont identiques. Dans le cas contraire, le réseau est dit *hétérogène*. L'approche *homogène* est la plus simple à mettre en oeuvre. L'approche *hétérogène* est la plus optimisée en terme de performance mais plus complexe à mettre en oeuvre du fait du mélange de types d'unité. Dans le domaine de l'embarqué, cette seconde approche est la plus pertinente. En effet, la surface étant limitée, il est souvent nécessaire de faire cohabiter des unités de calcul hétérogènes afin d'assurer la complémentarité en terme d'efficacité de calcul.

La mise en oeuvre d'un réseau sur puce, implique généralement un surcoût évident en surface par rapport à une interconnexion point-à-point. Ce surcoût est compensé par la flexibilité des communications parallèles et à une meilleure intégration des unités de calcul dans le système. Un très grand nombre de propositions de réseaux sur puce (NoC) a été étudié depuis plusieurs années [111]. De plus en plus de systèmes multi-processeurs sur puce ont alors adopté cette solution d'interconnexion dans des domaines d'application variés comme la télécommunication [112]. Les réseaux sur puce présentent également un intérêt croissant pour la réalisation de SoC dédié à la vision embarquée.

3.4 Réseaux sur puce utilisés en vision embarquée

3.4.1 Utilisation et Evaluation du NoC

Plus particulièrement, dans le domaine de la vision embarquée, plusieurs travaux ont été entrepris dans le cadre des réseaux sur puce. Les performances d'un NoC par rapport à d'autres méthodes d'interconnexion ont été montré pour différentes applications de vision. A titre d'exemple, Hilton dans [113] le montre en comparant les performances en temps et en surface d'un système d'interconnexion avec un bus partagé par rapport à sa proposition de réseau PNoC dans le cadre d'une application de binarisation d'image.

D'une manière générale, la mise en oeuvre d'un modèle de réseau sur puce nécessite une étude important dans l'exploration de différents paramètres du réseau comme la largeur de bus de données ou la taille d'un paquet de donnée. Cette étude appelée *Design Space Exploration* (DSE) est incontournable pour garantir des performances optimales selon des contraintes définies en terme de surface et de temps. Ce travail est d'autant plus important lorsque le modèle de réseau utilisé n'est pas adaptable au niveau architectural après son intégration sur puce. Des efforts en terme de compromis surface et temps, sont ainsi à considérer selon un ensemble d'applications visées. C'est le cas du modèle de réseau NoC Hermes [114] qui a été beaucoup étudié dans la littérature. Il a été récemment exploré par Fresse [115] afin de trouver le paramétrage le plus efficace dans le domaine d'application destiné à l'imagerie multi-spectrale. Cette exploration peut être éventuellement assistée par des outils dédiés afin de mapper efficacement une application. Par exemple, LeBeux propose un outil [116] pour explorer un NoC dans le but de mettre en oeuvre un algorithme de démosaiquage.

De ces travaux, nous pouvons conclure que plus un réseau NoC est figé au niveau architectural par la structure topologique ou la structure interne d'un routeur par exemple, et plus les efforts de développement doivent être portés en amont de sa mise en oeuvre au niveau de son paramétrage. Ce paramétrage est d'autant plus complexe que le nombre d'application à implémenter est croissant afin de garantir les performances.

A l'inverse, dans le contexte où le type d'application est défini, certains travaux ont proposé des réseaux sur puce dédiés en terme de topologie et de routage afin de garantir des performances optimales. Zhang [117] propose ainsi dans le cadre des algorithmes pour l'analyse d'image multispectrale, une topologie de réseau de type *Butterfly Fat Tree*. Nous pouvons encore citer un exemple de réalisation d'un MPSoC spécialisé dans la reconnaissance d'objet [118] utilisant une organisation mémoire dédiée et basé sur un réseau sur puce avec une topologie en étoile hiérarchique. Néanmoins, ce type de stratégie ne permet pas de changer facilement d'application du fait de la rigidité de la topologie et des unités de routage de données.

3.4.2 Adaptation fonctionnelle et architecturale du NoC

De l'étude précédente, nous voyons que la mise en oeuvre d'un NoC, dans un contexte de modification dynamique d'applications avec des contraintes de performance, est un problème complexe. En effet, si le réseau est figé architecturalement, seule l'adaptation en modifiant le chemin de donnée entre les routeurs, au niveau fonctionnelle, n'est réalisable en modifiant les stratégies de routage. Cependant, cette unique solution se révèle, soit insuffisante pour garantir les performances, ou soit très complexe, ce qui impacte fortement la surface nécessaire pour réaliser les unités de routage. Par conséquent, il est nécessaire d'étudier également l'*adaptation dynamique au niveau architecturale* du NoC pour pouvoir déployer efficacement une application.

En résumé, nous pouvons nous adapter généralement à deux niveaux dans le NoC :

1. au niveau *fonctionnel* avec l'algorithme de routage
2. au niveau *architectural* avec la structure du réseau

La première adaptation peut être mise en oeuvre dans un premier temps, en définissant une fonction de routage adaptative pour chaque noeud comme par exemple en utilisant un algorithme de type XY pour les réseaux à topologie mesh. Ainsi, le routage s'adapte

selon l'état du réseau. Ce type d'adaptation est fréquemment utilisé pour des réseaux qui sont tolérants aux fautes comme le QNoC [119, 120] ou dans le cadre d'intégration dynamique de modules de calcul comme le DyNoC [121, 122] afin de contourner des modules intégrés dans un réseau. Une autre approche plus directe consiste à modifier dynamiquement le routage dans chaque routeur en changeant les tables de routage comme utilisé par Bartic [123] ou Hilton dans le réseau PNoC [113]. Cependant, en utilisant uniquement cette adaptation fonctionnelle, les structures de communication restent encore trop rigides.

Le deuxième type d'adaptation consiste à travailler directement sur l'architecture. Un grand nombre de travaux proposent alors de modifier la topologie du réseau comme le Noc FLUX [10] ou d'autres solutions proposées par Gohringer [124, 125] ou Luedtke [126] afin de pouvoir garantir des performances optimales selon l'application. La stratégie de modifier la topologie du réseau permet de faciliter le parcours des données entre les unités de calcul selon l'application. Cependant, d'une part, cette stratégie n'implique pas forcément un acheminement efficace des données. D'autre part, elle permet difficilement d'obtenir un déterminisme en temps sur l'acheminement des données du fait du changement de topologie.

Il est donc nécessaire d'étudier conjointement une adaptation à la fois fonctionnelle et architecturale. Dans notre contexte, nous devons à la fois nous adapter aux applications et aux types de flux de données, tout en garantissant la performance.

3.5 Conclusion

A notre connaissance, il n'existe aucun système d'interconnexion de type réseau permettant d'acheminer en parallèle plusieurs flux de pixels avec la capacité d'auto-adapter le chemin de donnée entre les unités de calcul. Cette adaptation nécessite dans notre contexte d'être réalisée en fonction de la donnée et de l'application associée à la donnée.

Nous proposons ainsi de développer un nouveau *réseau de communication sur puce* (NoC) pour un SoC dédié à la vision.

Notre objectif étant d'obtenir un système d'interconnexion, paramétrable avant synthèse, disposant de la capacité d'auto-adapter son chemin de donnée. Cette adaptation

doit idéalement pouvoir s'effectuer de manière dynamique en fonction du flux de données pixélique qui nécessite d'être distingué entre les différents capteurs utilisés dans le sytème.

En particulier, comme décrit dans le chapitre 2 précédent, dans un contexte où des unités de calculs optimisées (IPs) ont été développées, des assemblages spécifiques comme des *pipelines* d'unités de calcul sont idéales pour traiter les images en flot de données. Dans ce but, nous proposons d'adapter directement au niveau architectural, le chemin de donnée interne du routeur afin de pouvoir garantir un ou plusieurs *pipelines* efficaces d'unités de calcul. Nous verrons, dans le chapitre suivant, que cette adaptation nécessite de mettre en oeuvre un routage spécifique afin d'organiser efficacement les flux de données traités en parallèle.

Chapitre 4

Proposition d'un NoC dynamiquement adaptable

4.1 Introduction

Dans ce chapitre, nous présentons, dans un premier temps, un nouveau modèle d'architecture de NoC, permettant de gérer dynamiquement plusieurs flux de données pixéliques en parallèle. Ces flux parallèles peuvent provenir de différents capteurs d'image ou peuvent être générés à partir d'un seul flux selon l'application.

La figure 4.1 illustre le problème à résoudre pour la conception du NoC avec la présence, par exemple, de trois capteurs d'image différents en entrée, deux afficheurs en sortie et d'un ensemble de PEs réutilisables pour plusieurs applications dans un SoC.

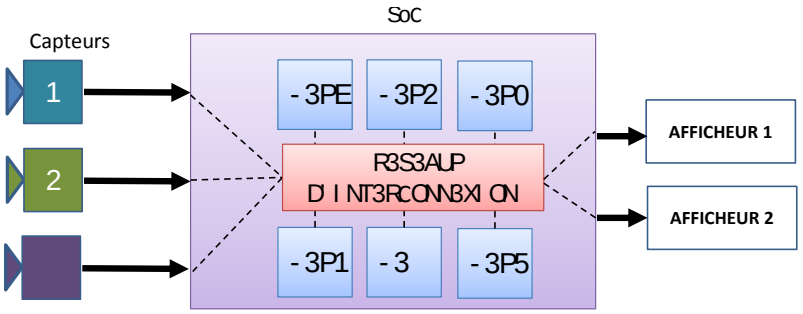


FIGURE 4.1: Système avec trois capteurs et deux afficheurs

Les capteurs en entrée peuvent être typiquement ceux présentés au chapitre 2, à savoir un capteur IR, BNL et couleur. Sur cette figure, nous voyons que toutes les images

en sortie de chaque capteur transitent dans le réseau d'interconnexion, dont le rôle est d'acheminer les images en sortie sur afficheur, après application de traitements définis. Ces traitements sont réalisés par l'association des différents PEs implantés dans le SoC. Le problème à résoudre se situe alors au niveau du réseau d'interconnexion qui doit être capable de gérer de multiples flux en parallèle et de les acheminer correctement vers les unités de calcul selon l'application, qui peut être modifiée dynamiquement.

Se trouve également posé le problème de la distinction de la provenance de chaque flux de données transitant dans le réseau. De ce fait, nous proposons une nouvelle méthode d'identification des paquets de données dans le réseau, particulièrement adaptée aux applications de vision embarquée.

En nous appuyant sur une nouvelle proposition d'architecture de NoC, nous décrivons par la suite une nouvelle méthode de routage des paquets. Il en résulte, au niveau architecturale, une modification dynamique du chemin de données interne au routeur. Ce routage est déterminé en fonction de l'identification des paquets et du séquencement des unités de calcul défini pour exécuter correctement une application.

4.2 Modèle d'architecture d'un NoC dynamiquement adaptable

4.2.1 Définition du flux de données

Nous utilisons le terme flux (du latin *fluxus*, écoulement) pour désigner un ensemble de données évoluant dans la même direction. Le terme "ensemble" étant défini, selon la théorie des ensembles, comme une collection d'éléments. Dans le domaine de la vision numérique, nous parlons de flux d'une image pour désigner une collection de valeurs de pixels. Un pixel (de l'anglais *picture element*) est une unité de surface permettant de mesurer une image numérique. À chaque pixel est associée une valeur d'une composante d'intensité (exprimée en bits) dans le cas d'une image monochrome en niveaux de gris, ou plusieurs valeurs de composantes dans le cas d'une image couleur selon l'espace colorimétrique choisi : Red Green Blue (RGB), YCrCb ou Hue Saturation Value (HSV) par exemple. Ainsi, un flux d'une image peut représenter soit un bloc d'image, une image complète ou une séquence d'image.

4.2.2 Choix d'une architecture orientée flux de données

La définition de notre modèle d'architecture se base sur le constat que la majorité des unités de calcul optimisées pour réaliser les opérateurs de traitement d'image, présentés au chapitre 2 en section 2.3.5, sont *orientées flux de données*.

Nous avons également remarqué dans ce même chapitre, que ces opérateurs sont réutilisables entre différentes applications de visualisation d'image, selon les capteurs utilisés. Ainsi, en supposant qu'un système dispose de l'ensemble des unités de calcul capables de réaliser toutes les applications requises, il serait idéal que le système d'interconnexion puisse adapter dynamiquement les liaisons entre les unités de calcul en fonction du type de donnée, distingué selon le capteur, et de l'application à implémenter.

Par ailleurs, nous avons vu que la majorité des architectures de vision embarquée privilégie, généralement, au mieux la possibilité de chaîner les unités de calcul sous forme d'un *pipeline* [127–129] afin d'obtenir un minimum de mémorisation, de pouvoir traiter directement en flot de données et ainsi maximiser la bande passante des canaux de communication avec une latence minimale.

Notre modèle d'architecture NoC tient compte de cette approche de conception, afin de pouvoir composer plusieurs pipelines d'unités de calcul, dans le but d'approcher les performances d'une architecture câblée spécifique mais sans la contrainte de liaisons point-à-point figées entre les unités.

4.2.3 Proposition du modèle d'architecture de NoC

Dans notre NoC proposé, un *noeud* correspond soit à un routeur unique ou soit à l'association d'une unité de calcul (PE) avec son routeur. Plus particulièrement, nous distinguons deux types de noeuds : les noeuds *maîtres* (M) et les noeud *esclaves* (E).

La figure 4.2 présente un exemple de mise en oeuvre du NoC dans le cadre de l'exemple précédent avec trois capteurs et deux afficheurs.

Sur cette figure, nous voyons que les capteurs et les afficheurs sont connectés au réseau au travers des routeurs maîtres (M), qui permettent de réaliser l'interface avec l'extérieur. Les PEs sont connectés au réseau au travers des routeurs esclaves (E) placés entre les noeuds maîtres.

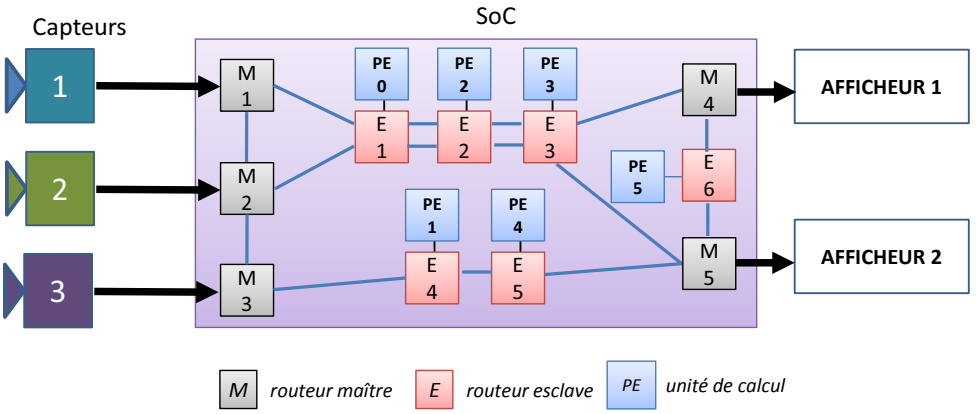


FIGURE 4.2: Principe de l'architecture du NoC proposé

D'un point de vue système, nous pouvons voir notre modèle comme un réseau *indirect* de routeurs maîtres qui sont reliés par plusieurs réseaux linéaires *directs* de routeurs esclaves.

Les communications principales du réseau s'effectuent entre les noeuds maîtres et les communications secondaires entre les noeuds esclaves. Un noeuds maître représente une source ou une destination principale pour un paquet de données. En particulier, un routeur maître est caractérisé par des interfaces avec le monde extérieur lui permettant d'acquérir un flux de données entrant et de sortir un flux de données traité. Dans notre modèle, les routeurs maîtres sont reliés ou non par un nombre fini de routeurs esclaves auxquels sont connectées des unités de calcul.

Le rôle d'un noeud esclave est d'analyser les paquets de données entrants et de rediriger ces paquets vers leur unité de calcul selon les besoins de l'application à réaliser. Un noeud esclave est ainsi capable de modifier le contenu des paquets de données transitant entre deux noeuds maîtres. Cette modification est effectuée en fonction du résultat fourni par l'unité de calcul connectée au routeur esclave.

La figure 4.3 illustre un exemple de liaison composée de trois routeurs esclaves, sur lesquels sont connectées les unités *PE0* à *PE2*, entre deux routeurs maîtres 0 et 1.

Dans cet exemple, les routeurs maîtres communiquent à travers deux canaux A et B entrants ou sortants. Un routeur esclave k est relié par deux liaisons *unidirectionnelles* entrantes $FW_{k-1}(A)$ et $FW_{k-1}(B)$, et deux liaisons sortantes $FW_k(A)$ et $FW_k(B)$. Le routeur maître 0 est ainsi le seul émetteur possible. Le flot de données entre les noeuds

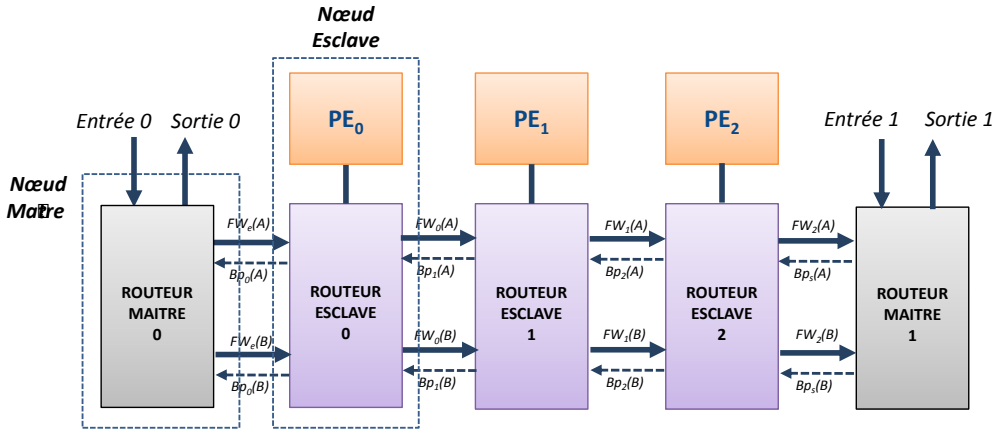


FIGURE 4.3: Réseau linéaire de routeurs esclaves entre deux noeuds maîtres

maîtres est contrôlé par des signaux d'acquittement $Bp(A)$ et $Bp(B)$ entre les routeurs esclaves.

Le chemin de données alloué pour faire transiter le flot peut être modifié dans un routeur esclave selon l'utilisation de l'unité de calcul et du séquencement des ces unités défini par l'application à réaliser.

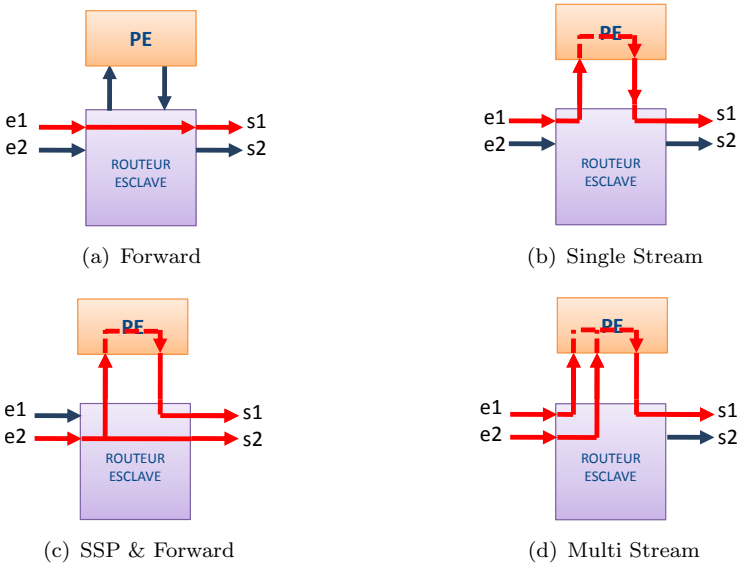
4.2.4 Modes de fonctionnement du routeur esclave

Un routeur esclave est capable de réaliser quatre modes de fonctionnement pour un ou plusieurs paquets entrants :

1. Forward (FWD)
2. Single Stream (SS)
3. Single Stream & Forward (SSF)
4. Multi Stream (MS)

La figure 4.4 illustre le chemin de données du paquet, surligné en rouge, défini pour chaque mode de fonctionnement, avec un exemple de routeur esclave constitué de deux canaux $e1$, $e2$ unidirectionnels entrants et deux canaux $s1$, $s2$ sortants.

Le mode *Forward* (FWD), illustré par la figure 4.4(a), consiste à rediriger un paquet de donnée entrant vers une sortie du routeur sans utilisation de l'unité de calcul (PE). Ce mode est utilisé dans deux cas : soit le PE n'est pas disponible ou soit le PE n'est pas capable de traiter la donnée selon une opération requise par l'application.



(a) Forward (b) Single Stream

(c) SSP & Forward (d) Multi Stream

FIGURE 4.4: Modes de fonctionnement du routeur esclave

Le mode *Single Stream* (SS), illustré par la figure 4.4(b), est utilisé si le paquet entrant peut être traité par le PE. Dans ce mode, le paquet est ainsi intégralement redirigé et traité par le PE.

Dans le mode *Single Stream & Forward* (SSF), illustré par la figure 4.4(c), le paquet de données est dupliqué : un exemplaire est envoyé directement vers une sortie sans aucun traitement et l'autre exemplaire est dirigé vers le PE. Les données, résultantes du traitement par le PE, sont systématiquement transmis par un autre canal de sortie disponible. Ce mode permet ainsi de transmettre deux flux de données en parallèle dans le réseau : le flux entrant et le flux traité par le PE.

Dans le mode *Multi Stream* (MS), illustré par la figure 4.4(d), le PE est capable de traiter N flux de données en parallèle ($N = 2$ dans notre exemple) pour produire une donnée sur un canal de sortie. Ce mode est particulièrement utile pour connecter des PEs particuliers qui sont capables de combiner en parallèle plusieurs flux de données. Il s'agit d'un cas typique pour l'implémentation d'un opérateur de fusion d'image, décrit au chapitre 2.

4.2.5 Règles de construction du réseau

Dans cette section, nous définissons des règles spécifiques pour la constructions de notre NoC afin de respecter les contraintes imposées par le modèle d'architecture de réseau proposé et la définition des modes de fonctionnement du routeur esclave. Ces règles sont à appliquer pour la spécification du NoC avant synthèse.

4.2.5.1 Règle n°1 : Latence maximum entre noeuds maîtres

Le réseau doit être construit en garantissant une latence maximum entre deux noeuds maîtres. Cette latence maximum correspond au nombre de cycles maximum autorisés pour la traversée d'une donnée entre deux routeurs. Elle est définie suivant le nombre de routeur esclaves entre les noeuds maîtres et des latences de calcul de chaque unité associée à un routeur esclave.

Soit L_{max} cette latence maximum autorisée entre deux noeuds maîtres qui sont reliés par k noeuds esclaves représentés par des couples routeur-unité de calcul (R_i, U_i). Soient L_{R_i} et L_{U_i} les latences respectives maximum pour les routeur esclaves et l'unité de calcul.

La construction d'une liaison entre deux noeuds maîtres doit ainsi respecter l'équation 4.1 suivante :

$$\sum_{i=0}^{k-1} max(L_{U_i}, L_{R_i}) < L_{max} \tag{4.1}$$

Dans un système de vision embarquée, cette latence peut être définie typiquement en fonction de la latence maximum autorisée entre l'entrée de la source image du capteur et l'affichage de cette image traitée.

4.2.5.2 Règle n°2 : Chemin de données pour traiter la donnée

Quelle que soit la topologie du réseau, le réseau doit être construit de manière à ce qu'il existe toujours un chemin de données capable de traiter complètement tout paquet de données provenant d'un noeud maître émetteur vers un noeud maître destinataire. Un nombre de liaisons unidirectionnelles doit alors être défini entre les routeurs pour établir tous les chemins de données nécessaires.

4.2.5.3 Règle n°3 : Parité des canaux entrants et sortants

Le nombre de ports de communication d'un routeur esclave est toujours pair. Les ports sont définis par n couples de port entrant et sortant. Un port entrant du routeur esclave est associé à un port sortant. Ainsi, en mode *Forward*, tout paquet entrant dans un port est automatiquement dirigé vers le port de sortie défini par le couple.

4.2.5.4 Règle n°4 : Nombre de canaux de communication minimum

Etant donné les modes de fonctionnement définis pour le routeur esclave, la construction du réseau n'est pertinente qu'à partir d'un minimum de *deux* canaux de communication entrants unidirectionnels par routeur esclave.

Nous pouvons noter que, parmi ces quatres règles de constructions définies, les règles n°2 et n°3 sont obligatoires pour que le réseau soit fonctionnel afin d'éviter le blocage des données. Les deux autres règles sont conseillées pour assurer les contraintes de performances.

4.2.6 Topologies du réseau

Différentes topologies peuvent être ainsi réalisées selon les règles de construction définies précédemment. Le choix topologique d'un réseau de communication présente un impact important sur les performances en temps et en surface d'une architecture de calcul.

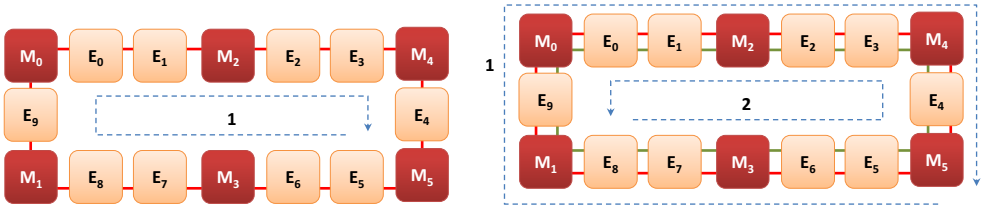
4.2.6.1 Topologie en anneau

La topologie en anneau est une solution qui est adaptée pour notre modèle d'architecture et pour nos applications orientées flot de données. Un principal avantage réside sur le faible degré de liberté des noeuds, ce qui permet ainsi de conserver une solution de routage simple par la linéarité de la structure.

Nous pouvons citer les travaux de Bourduas [130] qui ont montré l'efficacité d'un réseau de communication basé sur des anneaux unidirectionnels en terme de surface et de fréquence d'utilisation plus élevée, en raison de la simplicité de la structure. Une

étude comparative des topologies dans le cadre d'une implémentation FPGA, menée par Saldana [131], démontre également la pertinence de cette topologie.

Quelques exemples de topologie en anneau pour notre réseau sont présentés en figure 4.5. Les noeuds M et E représentent respectivement les noeuds maîtres et esclaves dans cet exemple.



(a) Anneau simple (b) Anneau double

FIGURE 4.5: Exemples de topologie en anneau et sens de circulation des données

La figure 4.5(a) présente une version simple avec un anneau unidirectionnel. Un exemple de sens de circulation (1) des données est représenté sous forme d'une flèche en pointillée sur la figure. Il s'agit d'une solution parmi les moins coûteuses en surface étant donné le faible nombre de ports pour chaque routeur. Cependant, cette solution ne respecte pas une des règles de construction et limite les modes d'exécution des routeurs esclaves car elle ne permet pas d'effectuer des traitements en parallèle.

La figure 4.5(b) propose un autre exemple avec deux anneaux unidirectionnels, constituant un anneau double, dans les deux sens de circulation (1) et (2), représentés par des flèches en pointillées. Cette dernière solution présente un avantage de pouvoir faire circuler les paquets dans les deux sens de circulation. Elle permet ainsi à un paquet entrant d'atteindre plus rapidement un routeur de destination sans devoir effectuer un tour complet dans le pire cas.

Plus généralement, un grand nombre d'architecture multiprocesseurs ont opté pour la topologie en anneau, en particulier pour des applications en flot de données. Dans la famille généraliste, nous pouvons citer par exemple le processeur *Cell* [132] conçu par IBM, Toshiba et Sony. Dans le domaine du traitement de signal audio, nous pouvons citer les processeurs X-Fi [133] de la société Creative. Plusieurs architectures spécialisées ont également exploité la topologie anneau dans le domaine du traitement d'image [17, 34] et du signal comme l'architecture *Systolic Ring* [134, 135].

Nous pouvons également remarquer que la solution en anneau est particulièrement bien adaptée pour des applications utilisant des opérations itératives. Une même donnée peut ainsi être traitée de manière récursive par des rebouclages successifs dans l'anneau. Cette méthode permet ainsi de minimiser la surface de calcul en multiplexant temporellement les opérations.

Avec une topologie en anneau, l'efficacité du calcul est ainsi dépendante de la profondeur du pipeline réalisable avec une même unité de calcul. Cette profondeur peut être compensée par des rebouclages de la donnée à traiter, reproduisant ainsi un mode de calcul que nous appelons *pipeline virtuel*.

Prenons l'exemple d'une application réalisée avec une suite d'opérations séquentielles comme illustrée par la figure 4.6 avec $2n$ opérations successives. Dans cet exemple, cette suite peut être découpée en un groupe de deux opérations (OP1 et OP2) qui est itéré en n fois.

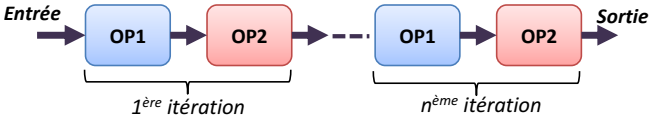


FIGURE 4.6: Exemple d'application composée de n itérations

Supposons à présent que les opérations OP1 et OP2 sont respectivement réalisées par les unités de calcul PE1 et PE2. En configurant un réseau capable d'exécuter en pipeline deux groupes d'opération, comme présenté en figure 4.7, l'application complète peut être ainsi réalisée en rebouclant $\frac{n}{2}$ fois dans l'anneau chaque donnée entrante.

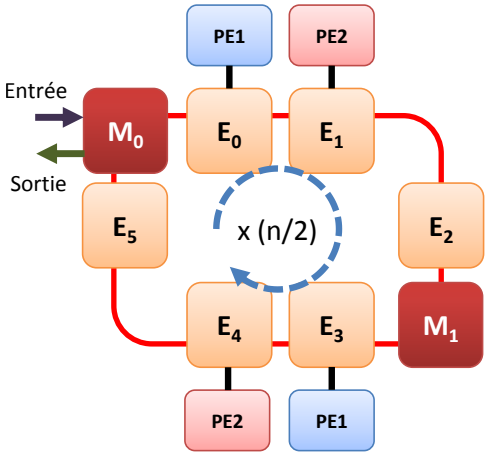


FIGURE 4.7: Exécution d'un pipeline virtuel en deux rebouclages des données

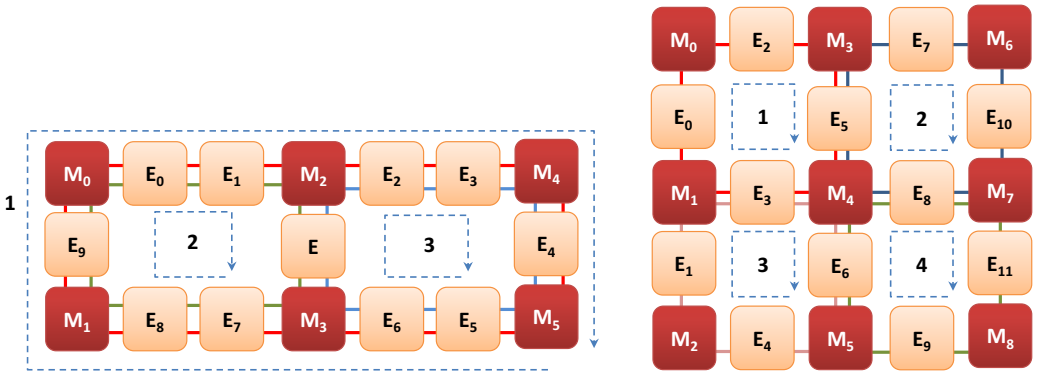
De façon évidente, comparée à une solution complètement pipelinée spatialement, cette méthode temporelle de calcul est moins avantageuse en terme de latence à une fréquence de fonctionnement égale. Cette méthode implique également de mettre en oeuvre une gestion des flux de données pour éviter des collisions lors du rebouclage. Cette gestion peut être assurée par un découpage adéquat des données, avec un séquencement des transmissions, et la mise en oeuvre d'un système de mémorisation permettant de bufferriser les données, pour éviter les collisions lors des bouclages. Cependant, dans le cadre d'application moins critiques en terme de temps et pour des unités de calcul de surface importante, cette solution de rebouclage reste pertinente [20].

4.2.6.2 Extension de la topologie en anneau

Une topologie en anneau peut être étendue vers des topologies plus complexes avec une structure se rapprochant de la topologie grille (*mesh*). La topologie grille est celle qui est la plus utilisée pour les réseaux de communication sur puce, en raison de sa simplicité de conception en deux dimensions et de la scalabilité en nombre d'unités de routage.

Cette topologie est applicable dans le cadre de notre proposition, mais il est nécessaire de tenir compte des règles de construction. Il s'agit en particulier de l'unidirectionnalité des canaux et de la parité entre un port d'entrée et un port de sortie.

Quelques exemples de topologies en grille réalisables sont présentées par les figures 4.8(a) et 4.8(b).



(a) Anneaux imbriqués (b) Anneaux découplés

FIGURE 4.8: Autres exemples de topologie

Le premier exemple, en figure 4.8(a), présente une topologie réalisée avec des anneaux imbriqués. Dans cet exemple, nous pouvons voir deux anneaux unidirectionnels (2) et (3) qui sont restreints à un partie du réseau. Ces anneaux sont imbriqués dans un anneau principale (1) permettant l'accès à tous les routeurs du réseau.

Dans le second exemple, en figure 4.8(b), les anneaux (1), (2), (3) et (4) sont complètement découplés. Le seul moyen de passer d'un anneau à un autre est d'effectuer des changements consécutifs au niveau des routeurs maîtres (M).

Chacune des solutions possède ses avantages et ses inconvénients en latence de routage, avec des algorithmes spécifiques à la topologie. Le travail d'exploration topologique du réseau, à partir de notre modèle d'architecture, dépasse le cadre de notre étude dans cette thèse.

4.3 Contrôle du mode de fonctionnement des routeurs

4.3.1 Contrôle centralisé et distribué

Il existe deux types de contrôle : un contrôle *centralisé* ou un contrôle *distribué*.

Dans le cas d'un contrôle *centralisé*, la prise de décision est réalisée au niveau système. Il peut s'agir d'un module central qui s'occupe du contrôle global des ressources (mémoires et unités de calcul). Dans le cas d'une application orientée flux, une architecture classique consiste à placer un module de commande amont avec une topologie de contrôle en étoile vers chaque module de calcul agencés sous la forme d'un pipeline, comme illustré par la figure 4.9.

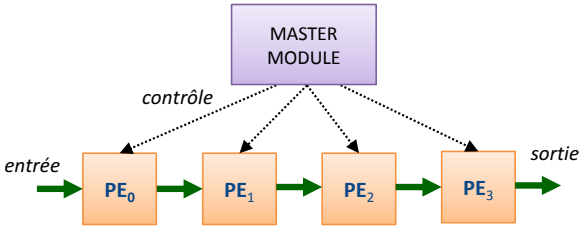


FIGURE 4.9: Exemple de contrôle centralisé

Dans le cas d'un contrôle *distribué*, la prise de décision est effectuée par plusieurs unités indépendantes ou en intéraction. Le contrôle est distribué sur l'ensemble des unités qui prennent des décisions selon leurs interactions mutuelles (figure 4.10).

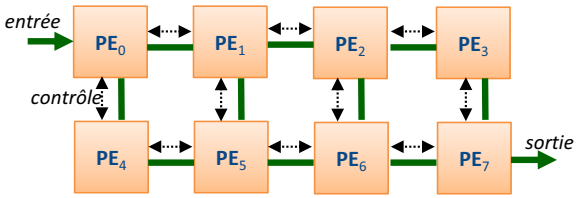


FIGURE 4.10: Exemple de contrôle distribué

Un contrôle *distribué* permet d'avoir un système de contrôle plus réactif, avec une prise de décision plus rapide et locale, par rapport un contrôle centralisé. Il nécessite néanmoins d'ajouter des informations supplémentaires dans les paquets de données ou des signaux supplémentaires afin que les unités puissent se communiquer mutuellement .

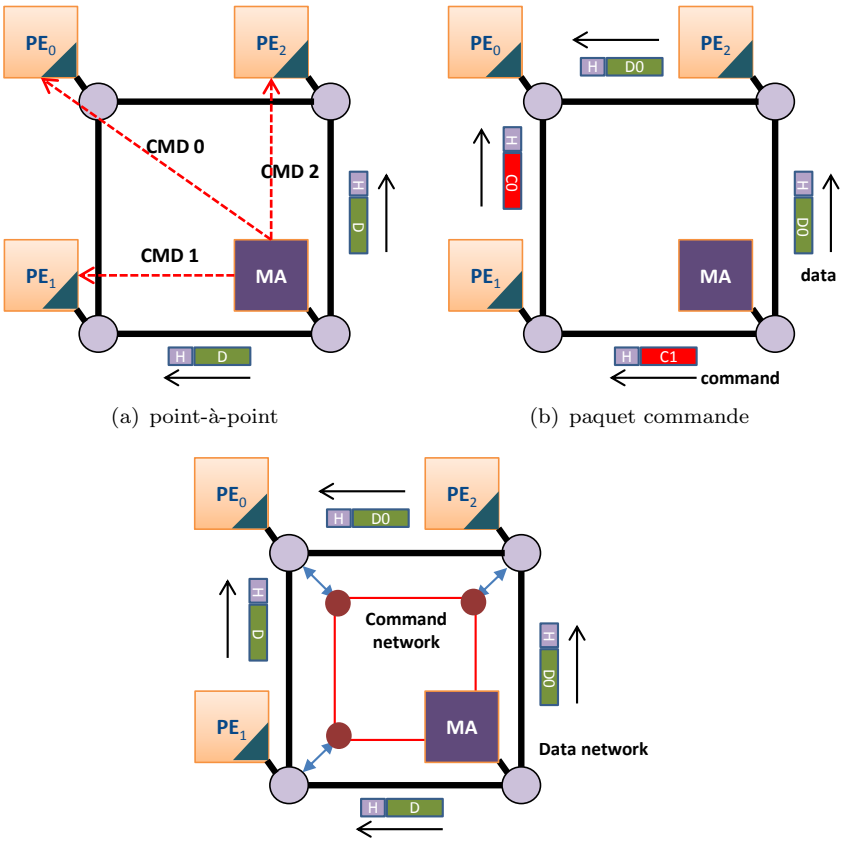
Le contrôle centralisé permet généralement d'avoir une meilleure maîtrise du chemin global du flot de données, alors que dans une solution décentralisée, le flot de donnée est géré par plusieurs unités de contrôle et peut suivre différents chemin de données pour acheminer l'information. Cependant, le système de contrôle centralisé est plus lent car il doit superviser un ensemble plus ou moins important d'unités et prendre des décisions selon l'état de chaque élément. Si le seul module de contrôle n'est plus fonctionnel alors tout le système n'est plus opérationnel. Il peut cependant convenir pour un système avec un faible nombre d'unités de calcul.

4.3.2 Implantation du contrôle des routeurs esclaves

Les modes de fonctionnement définis pour le routeur esclave permettent d'adapter le chemin de données entre différentes unités de calcul de manière à implémenter efficacement des applications orientées flot de données. Ces modes permettent entre autres de construire plusieurs pipelines d'unités de calcul en parallèle. Se trouve à présent posé le problème de la transmission des *commandes d'adaptation*, permettant de contrôler les modes de fonctionnement de chaque routeur esclave.

Une première solution consiste à définir un contrôle centralisé avec un contrôleur global maître, illustré par MA sur la figure 4.11, dont le rôle sera de superviser, de séquencer et de commander les routeurs esclaves. Ces routeurs peuvent être connectés au contrôleur par des liaisons directes point-à-point ou indirectes.

Une solution par *liaisons directes*, comme illustré par la figure 4.11(a) avec les liaisons de commande CMD0, CMD1 et CMD2, est la plus simple mais n'est naturellement pas



(a) point-à-point (b) paquet commande

(c) réseau dédié

FIGURE 4.11: Méthode de communication des commandes

adaptée pour un système évoluant vers un grand nombre d'unités de calcul. Cette solution nécessite en outre, des modifications permanentes du contrôleur global. L'utilisation de *liaisons indirectes*, avec des méthodes d'interconnexion par des bus ou un réseau de communication, autorise une plus grande intégration de noeuds esclaves mais elle introduit généralement une latence de communication qu'il est nécessaire de minimiser.

La figure 4.11(b) illustre un exemple avec des paquets de commandes, avec un en-tête `[H]` et une donnée `[C]`, et des paquets de données, avec un en-tête `[H]` et une donnée `[D]`, envoyés successivement. Selon la méthode d'interconnexion, le nombre de routeurs esclaves et les transferts de données entre unité maître-esclaves, il apparaît généralement un goulot d'étranglement des données au niveau du contrôleur global.

Une méthode pour résoudre partiellement ce problème est de définir des *liaisons dédiées uniquement au contrôle* et aux commandes, comme illustré par la figure 4.11(c), mais elle implique un surcoût en surface [100, 117].

Afin d'optimiser la communication du contrôle, Clermidy [136] a proposé récemment de distribuer le contexte de configuration sur plusieurs noeuds dans le réseau et de mettre en oeuvre un protocole permettant de modifier localement les unités de calcul de façon décentralisée. Cette solution nécessite cependant une complexité et une consommation en surface plus importante dans la couche d'interface réseau pour chaque noeud. De manière évidente, l'efficacité de cette proposition est dépendante du nombre et de la position des unités de configuration dans le réseau.

Lorsque la cible technologique le permet, une alternative à cette première méthode consiste à tirer profit de la reconfiguration dynamique pour configurer les multiplexeurs et ainsi modifier les connexions entre les ports d'entrée-sortie dans le routeur. En guise d'exemple, cette solution a été mise en oeuvre par Braun [137] dans le cadre d'un NoC basé sur des routeurs adaptables implémentés sur une cible FPGA. Cette solution est pertinente mais est trop fortement dépendante du composant technologique, qui est un FPGA particulier dans cet exemple, et nécessite de connaître physiquement et exactement le positionnement spatial des modifications.

4.3.3 Proposition de contrôle

Notre proposition consiste à *combiner dans un paquet à la fois les données à traiter et les instructions à appliquer sur la donnée* en fonction de l'application à implémenter. Ces instructions décrivent un séquencement d'opérations sur la donnée pouvant être réalisé par des unités de calculs dans le réseau. Suivant ces instructions, les routeurs esclaves adaptent dynamiquement le chemin de données selon les modes de fonctionnement définis. Dans notre réseau, le contrôle est ainsi partiellement *distribué* car le routeur maître n'a pas de connaissance sur la position des routeurs esclaves. Les routeurs esclaves organisent dynamiquement le chemin de données en fonction des instructions, de la donnée et de l'occupation de l'unité de calcul.

La figure 4.12 illustre des paquets de données de taille plus importante, contenant à la fois des informations d'adressage `[H]`, de commandes `[C]` et de données `[D]`.

Pour notre réseau, ces commandes d'adaptation du chemin de données sont spécifiées sous forme de groupes d'instructions. Ces groupes d'instructions sont définies à partir de l'application en imposant un séquencement d'opérations requises sur la donnée. Ces

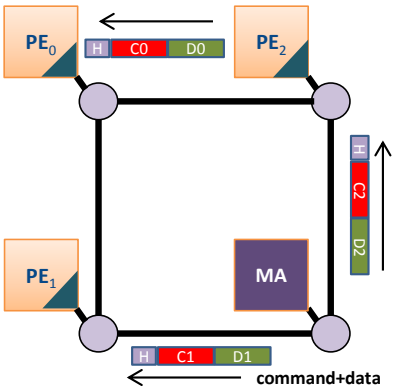


FIGURE 4.12: Méthode de communication des commandes avec la donnée à traiter

instructions sont intégrées directement dans l'en-tête du paquet de données. Nous détaillons le contenu de ces instructions, dans la section suivante, à partir de la définition de la structure du paquet de données.

4.4 Description des paquets de données

4.4.1 Attributs d'une image

Comme décrit au chapitre 2, une image nécessite d'être traitée afin de corriger les défauts de l'image, faire ressortir des zones particulières ou simplement améliorer l'affichage de l'image en sortie. Pour un système à multiples sources d'images et capable de réaliser différentes applications, un premier problème se pose concernant l'identification de la provenance et le type de flux de données que l'on souhaite faire traiter par un ensemble d'unités de calcul dans le réseau.

Ce problème n'apparaît pas pour une architecture câblée dont le système d'interconnexion repose sur des liaisons point-à-point. Dans ce cas de figure, les chemins de données sont exclusifs et préalablement définis avec des liaisons figées entre les unités de calcul. Par contre, dans un contexte de réutilisation des PEs et d'adaptation dynamique des liaisons de communication pour différentes applications, l'identification d'une image devient critique et importante. Il s'agit d'un besoin d'identification à la fois spatiale, pour distinguer les sources d'images, et temporelle, pour distinguer des images successives provenant d'une même source.

Dans notre contexte d'application, certaines opérations de type $2D+t$ (spatial et temporel) nécessitent effectivement d'identifier le positionnement dans le temps des différentes trames entrantes. Cet indice de temps permet ainsi de déterminer le Δ_t entre une trame stocké en mémoire et une trame en cours de traitement. Nous définissons ce Δ_t comme étant la différence relative en nombre de trames entre une trame en cours de traitement et une autre trame stockée en mémoire, pour une même position dans une chaîne de traitement. Cette information facilite grandement l'adaptation du système avec des applications utilisant des opérateurs temporels.

La figure 4.13 illustre un exemple d'une application temporelle utilisant plusieurs images à différents indices temporels. Dans cet exemple, le PE 0 combine en parallèle les images de la source 1 avec un Δ_t de 3, et le PE 1 combine celles de la source 2 avec un Δ_t de 1. Les images de $t-1$ à $t-3$ doivent ainsi être stockées dans une mémoire afin de pouvoir les relire selon les besoins des unités de calcul.

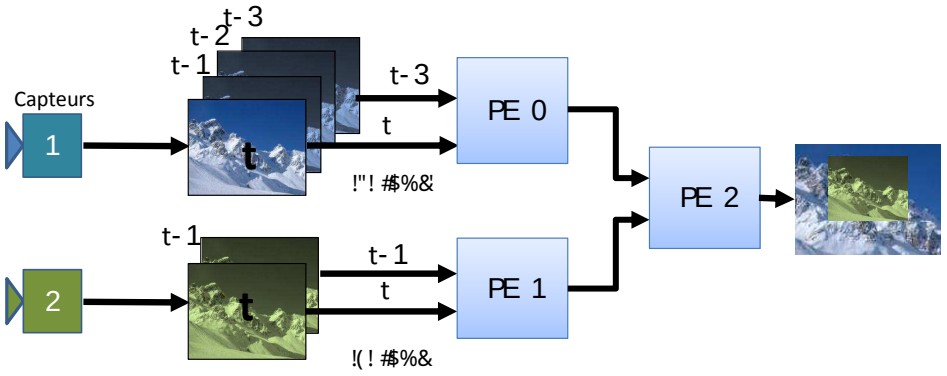


FIGURE 4.13: Exemple d'une application temporelle multi-sources

Le principe étant de définir une "carte d'identité" de l'image dont la précision dépend d'un certain nombre d'attributs. Nous appellons *attribut* une information nécessaire à l'identification des images dans le réseau. Par exemple, un opérateur de fusion de deux trames peut nécessiter à un instant t, une trame de la source du capteur infrarouge 1, à un Δ_t de 2, qui a été traitée par une opération de réhaussement de contraste ; puis à un instant $t1$ une trame de la même source mais avec un Δ_t de 1.

Dans le contexte de notre réseau, les messages sont principalement des trames d'image complètes ou des blocs d'image. Plusieurs paquets circulent alors en parallèle et transportent différents types de données comme les pixels d'une image. Ces paquets de données sont identifiés par leur en-tête.

Pour un réseau donné, le contenu de l'en-tête d'un paquet est généralement défini par le nombre et le type de données à traiter. Un en-tête peut contenir différents champs dédiés et il est défini selon les caractéristiques du réseau et les applications à implémenter. Ainsi, le contenu de l'en-tête d'un paquet peut se réduire simplement à un champ dédié à l'adresse du noeud de destination avec optionnellement des champs d'identification. Ces champs peuvent caractériser le *type* de donnée dans le paquet comme dans le cas du réseau QNoC [120] ou peuvent indiquer la précision des données [117]. Selon la technique d'aiguillage et l'algorithme de routage, l'en-tête peut contenir également un champ dédié au port de sortie défini dynamiquement à chaque passage dans un routeur, comme le propose par exemple Kavaldjiev dans le VCNoC [15], qui exploite le principe de *canaux virtuels* dans le conception du routeur.

Dans une situation mono-capteur et mono-application, où nous avons connaissance du type de source, un contrôleur global connecté au réseau reste suffisant, comme proposé par Zhang [117] dont le réseau contient une unité de contrôle qui se charge de gérer de multiples sources de données en mémoire vers plusieurs unités de calcul en parallèle. Sans contrôleur global, il est nécessaire d'utiliser au minimum un *algorithme de routage* permettant de garantir l'*ordre* des trames. Si il n'y a pas de garantie de l'ordre, une autre alternative serait de réorganiser l'ordre directement par une couche d'interface réseau spécifique de l'unité de calcul. Cette dernière solution implique un surcoût évident en surface.

Dans notre contexte multi-applicatif, ces stratégies ne peuvent suffire car le système doit s'adapter dynamiquement suivant l'application. Il est ainsi impératif, sans contrôle centralisé, d'enrichir l'identification des paquets dans notre réseau.

Nous pouvons conclure qu'une image nécessite au minimum trois attributs pour être identifiable dans notre réseau :

1. un attribut, nommé *id*, permettant de distinguer la source de l'image
2. un attribut, nommé *ts*, permettant d'identifier temporellement les images d'une même source
3. un attribut, nommé *pe*, permettant d'identifier la dernière opération effectuée sur la donnée

D'autres attributs utiles peuvent être également définis comme la *résolution* de l'image transmise dans le réseau.

4.4.2 Structure d'un paquet de données d'une image

La structure d'un paquet dans un réseau est illustrée par la figure 4.14. Elle correspond à l'association d'une donnée utile, aussi appelée *charge*, constituée de p flits, avec un en-tête composé de k flits.

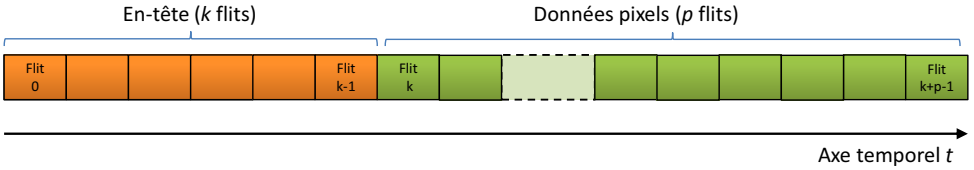


FIGURE 4.14: Structure d'un paquet de taille $k + p$ flits

Dans le cadre de nos applications pour la vision, cette charge représente un groupe de pixels provenant d'une image pour la majorité des paquets circulant dans le réseau. Soit une charge représentant un bloc d'image de résolution $M \times N$ pixels. Soient g la taille en bits des pixels et f la taille en bits d'un flit de données, définie pour le réseau. Le nombre de flits p sera ainsi déterminé par l'équation 4.2.

$$p = \lceil (M \times N \times g) \pmod{f} \rceil \tag{4.2}$$

Dans notre réseau, nous autorisons une charge pouvant atteindre la taille d'une image complète.

Contrairement à la taille variable de la charge, la taille de l'en-tête est *fixe*. Dans la plupart des réseaux sur puce, cette taille se réduit généralement à un seul flit ($k = 1$). Cette taille réduite permet de minimiser la latence du décodage de l'en-tête et de transmission des paquets. Dans ce cas précis, la latence de décodage peut se réduire à un seul cycle d'horloge si la taille du flit est égale à celle du phit.

Suite à notre besoin d'identification de la provenance des paquets, l'en-tête de nos paquets de données ne peut se réduire à un seul flit de données, si nous souhaitons une taille de bus de l'ordre de 16 ou 32 bits pour chaque liaison entre les routeurs.

Comme illustré par la figure 4.15, la structure d'un en-tête de paquet de données est divisée en trois parties principales : une partie indiquant la taille du paquet (*packet size*) de taille S_p, une partie pour décrire les instructions associées aux données transportées (*instructions*) de taille S_i et une partie réservée aux attributs caractérisant les données (*attributes*) de taille S_a. Ces trois parties peuvent être de taille variable et sont composées de un ou plusieurs flits de taille f. L'en-tête est alors délimitée par des flits de taille S_t, indiquant le début (*start*) et la fin (*end*) de sa description.

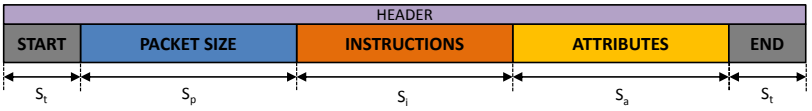


FIGURE 4.15: Structure de l'en-tête d'un paquet de données

Ainsi, la taille de l'en-tête S_H sera calculée suivant l'équation 4.3.

$$S_H = (2 \times S_t) + S_p + S_i + S_a \tag{4.3}$$

La partie indiquant la taille du paquet de données est impérative car celui-ci ne contient pas de flit de queue (*tail*) pour délimiter la fin du paquet. Ainsi, cette information peut être, soit indiquée directement en explicitant le nombre de pixels total transporté, ou soit de manière indirecte, en indiquant la résolution (en X et Y) du bloc de pixels contenue dans le paquet.

La partie détaillant les attributs contient principalement les informations caractérisant les données du paquet (id, ts, pe) et les informations d'adressage entre les noeuds maîtres source-destination.

La partie réservée aux instructions peut contenir plusieurs instructions selon la taille s_n définie pour une instruction telle que $S_i = k * s_n$ (avec $k \in \mathbb{N}$).

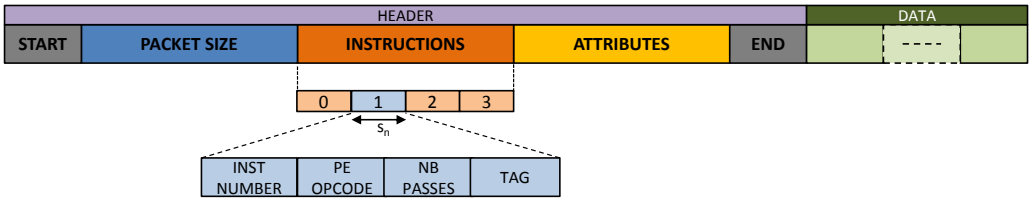


FIGURE 4.16: Partie réservée aux instructions dans l'en-tête (k=4)

La description et le séquencement des opérations sur la donnée transportée sont fixés par les instructions. Comme illustré par la figure 4.16, Une instruction est organisée en quatre champs :

1. *INST NUMBER* : Un champ correspondant au numéro de l'instruction à exécuter afin d'assurer la cohérence du séquencement des opérations sur la donnée transportée. Ce champ est particulièrement utile pour identifier les paquets de données appartenant à l'exécution d'une même instruction et qui doivent être traités en parallèle par le PE.
2. *PE OPCODE* : Un champ permettant d'identifier le type d'opération nécessaire pour traiter le paquet.
3. *NB PASSES* : Un champ qui fixe le nombre d'itération de l'opération sur la donnée. Du point de vue du code instruction complet décrivant l'application, ce champ permet de réduire les redondances d'opérations.
4. *TAG* : Un champ permettant d'indiquer le séquencement des opérations. Ce champ peut soit prendre une valeur ($TAG = PAR$) pour indiquer que l'instruction suivante peut être exécutée en parallèle ou soit le contraire ($TAG = /PAR$) pour indiquer que l'instruction suivante est exécutée séquentiellement. Dans le premier cas, un paquet de donnée traité peut être transmis en parallèle à un routeur voisin pour réaliser une autre opération. Il y aura donc édition de deux en-têtes différentes : une pour le paquet traité et une autre pour le paquet transmis. Dans le deuxième cas, le paquet doit être traité avant d'être transmis.

L'association de plusieurs instructions dans un en-tête permet de décrire les modes d'exécution de type *séquentiel-pipeline*, *parallèle* et *pipeline-parallèle* utilisés dans les différents opérateurs de traitement d'image décrits au chapitre 2.

Quelques exemples de code à quatre instructions par en-tête (k =4) sont présentés en figure 4.17.

Dans ces exemples, nous considérons que $\forall i \in \mathbb{N}$, l'unité PEi réalise l'opération OPi. Dans le premier cas, nous avons un traitement de type séquentiel entre trois opérations OP0, OP1 et OP2. Comme illustré en figure 4.17, ces opérations peuvent être pipelinées dans le réseau. Ce séquencement résulte du TAG de non parallélisme ($/PAR$) entre chaque opération. A l'inverse dans le deuxième exemple, nous souhaitons exécuter en

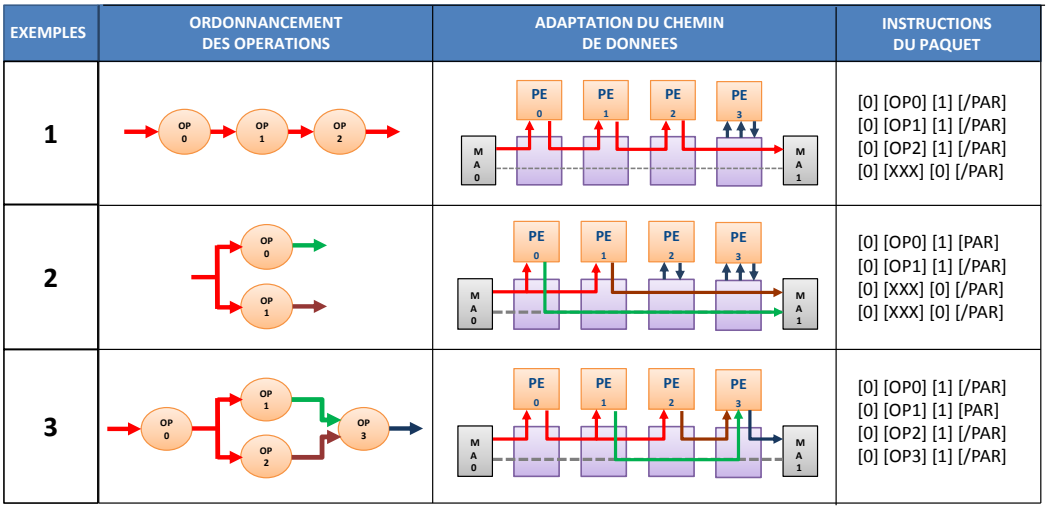


FIGURE 4.17: Exemples de code instruction associé à un ordonnancement des opérations sur les PEs

parallèle les opérations OP0 et OP1. Le *TAG* (*PAR*) associé à l'opération OP0 indique ce parallélisme qui autorise au routeur associé à PE0 de diffuser le paquet de donnée en parallèle du traitement par PE0. Le troisième exemple illustre une combinaison des deux exemples précédents pour réaliser à la fois un traitement parallèle puis séquentiel en combinant deux paquets traités par PE1 et PE2 vers PE3.

Nous pouvons remarquer que la structure du paquet ne contient pas de flit de queue (tail). Ce flit est généralement utilisé pour déterminer la fin du paquet. Dans notre cas, il n'est pas utile car la taille de notre en-tête k est fixe, et celui-ci contient déjà une information sur le nombre de pixels transporté dans le paquet.

Les routeurs esclaves sont majoritaires dans notre réseau. Etant donné que ce réseau doit être capable de gérer plusieurs flux de données en parallèle avec l'activation de différents modes de fonctionnement du routeur esclave, les besoins en terme de mémorisation risquent d'être importants dans chaque routeur et doivent être minimisés. De plus, ce problème est d'autant plus important que nous autorisons, dans notre modèle, des paquets de grande taille, équivalent à plusieurs lignes d'une image. En effet, étant donné que la taille de l'en-tête d'un paquet s'étend sur plusieurs flits, et que les opérations appliquées peuvent être pipelinées, le maintien de l'adaptation d'un chemin de données doit s'établir pour un minimum de pixels, afin de minimiser la latence de l'adaptation.

La taille des buffers dans un routeur est définie selon la *méthode de routage* utilisée pour acheminer les données dans le réseau. Elle est donc fortement dépendante de la *technique*

d'aiguillage utilisée (*switching technique*), du *contrôle de flot* de donnée choisi (*flow control*) et de l'*algorithme de routage* qui est plus ou moins efficace selon la topologie globale du réseau.

La topologie linéaire des routeurs esclaves entre deux routeurs maîtres bénéficie d'un faible degré de liberté à chaque noeud. Ce faible degré de liberté permet de réduire la complexité du routage à chaque noeud esclave. Nous étudions ainsi dans la section suivante la méthode de routage la plus adaptée à notre modèle d'architecture de réseau.

4.5 Routage dynamique des paquets de données

4.5.1 Technique d'aiguillage dans un réseau sur puce

4.5.1.1 Définition

La technique d'aiguillage détermine la manière dont les messages, paquets et flits traversent les routeurs d'un réseau de communication. Plus précisément, la technique d'aiguillage permet de définir comment les ressources du réseau (bande passante des canaux de communication, capacité mémoire des routeurs, etc.) sont allouées pour que les paquets puissent traverser le réseau. La méthode idéale consiste donc à allouer les ressources de façon à atteindre le maximum de bande passante tout en délivrant les paquets avec la plus faible latence possible. Dans ce but, plusieurs stratégies existent et sont regroupées selon deux catégories : les techniques *avec buffer* et les techniques *sans buffer*.

Les techniques sans buffer sont les plus simples à mettre en oeuvre car elles ne nécessitent aucun stockage de donnée. Ainsi, il n'est pas possible de bloquer un paquet dans le réseau car il n'y aucun emplacement de stockage dans les routeurs. A l'inverse, l'usage de buffer permet d'être plus souple dans la gestion du flot de donnée car les routeurs peuvent stocker des paquets en attente ce qui permet de partager temporellement les chemin de données disponibles dans le réseau.

Dans le domaine des réseaux sur puce, il existe deux principales techniques d'aiguillage : technique par aiguillage de circuit, sans buffer, appelée *circuit switching*, et par aiguillage de paquets, avec buffer, appelée *packet switching*. Ces techniques fixent la granularité du transfert de données.

Prenons un exemple basé sur notre réseau, illustré par la figure 4.18, avec deux routeurs maîtres réliés par trois routeurs esclaves. Dans cet exemple, les routeurs esclaves sont reliés entre eux par deux liaisons unidirectionnelles $FW_k(A)$ et $FW_k(B)$, avec $k \in \mathbb{N}$.

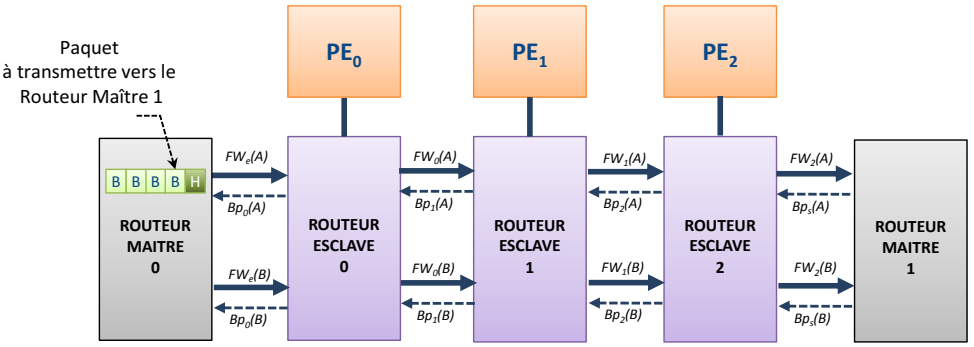


FIGURE 4.18: Noeuds maîtres séparés par trois noeuds esclaves

A partir de cet exemple, nous allons étudier l'utilisation de différentes techniques d'aiguillage appliquées au routeur esclave. Nous nous positionnons dans le contexte où le routeur maître 0 souhaite transmettre un paquet de donnée, dont la charge est équivalente à quatre flits, vers le routeur maître 1. Nous considérons que dans notre réseau, la taille du flit est égale à celle du phit.

4.5.1.2 Circuit switching

La technique de circuit switching [14] consiste à réserver le chemin de données physique (lien physique et routeurs) entre le noeud émetteur (source) et le noeud récepteur (destination) avant de pouvoir transmettre un ou plusieurs paquets de données. Cette réservation est effectuée par des signaux de requête-acquittement entre les routeurs. Nous pouvons citer le réseau SoCbus [138], proposé par Wiklund, qui implémente cette technique.

Dans notre réseau, une requête peut être transmise par un flit R de réservation spécifique. Ce flit contient l'adresse du noeud émetteur et celle du noeud récepteur. Les routeurs successifs valident cette requête par un signal d'acquittement correspondant aux signaux Bp_x pour notre exemple 4.18 et indiqué par la valeur A sur la figure 4.19.

Le chronogramme 4.19 illustre l'exemple de transmission d'un paquet à 4 flits de données du routeur maître 0 vers le routeur maître 1. Ce chronogramme montre le cheminement du paquet entre le routeur esclave 0 au routeur maître 1.

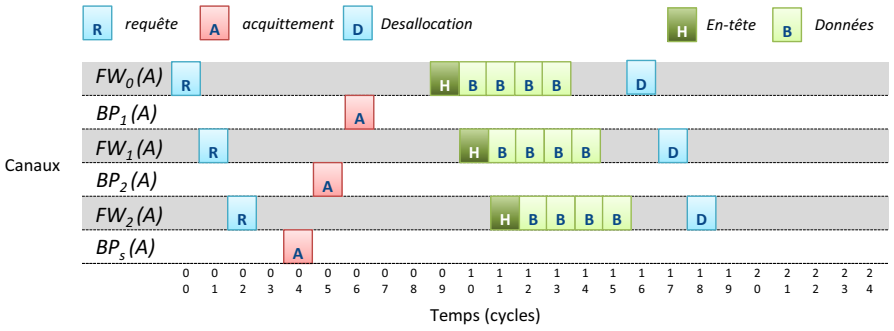


FIGURE 4.19: Chronogramme de la méthode Circuit Switching

Nous observons dans ce chronogramme une première phase de réservation de chemin de données, appelée *setup time*. Cette phase se décompose par la transmission du flit de réservation R et la propagation du signal d'acquittement A vers le noeud émetteur. Une fois ce signal reçu par le routeur maître émetteur, celui-ci peut alors transmettre son message en intégralité, sous forme de plusieurs paquets, sans risque de collision. La transmission se termine par l'envoi d'un flit D de désallocation afin de libérer progressivement les routeurs réservés sur le chemin de données, durant un temps appelé *tear-down time*.

Les avantages de cette technique sont une *bande passante maximale* (ressources réservées) et une *faible latence* de transmission entre le noeud source et destination.

Cependant, comme nous pouvons observer dans le chronogramme, cette technique implique une durée d'occupation importante du chemin de données. Elle prend en compte un temps de mise en oeuvre d'une transmission, imposée par la réservation et la désallocation du chemin de données.

4.5.1.3 Packet switching

Contrairement à la technique de circuit switching, les paquets sont transmis directement dans le réseau et le chemin de données est déterminé *dynamiquement* à chaque noeud de routage rencontré, selon un algorithme de routage défini. Il n'y a donc plus de latence occasionnée par un besoin de réservation de chemin.

Il existe dans la littérature différentes méthodes de packet switching [14] : le *Store and Forward* (SAF), le *Virtual Cut Through* (VCT), le *Wormhole Switching* (WS) et une variante utilisant des *canaux virtuels*.

Dans le cas d'une technique de type *Store and Forward* (SAF), un paquet ne pourra être transmis d'un routeur à un routeur voisin si ce dernier possède la capacité de stocker entièrement un paquet à transmettre. Ainsi, la taille des buffers des routeurs utilisés doit être au moins égale à la taille d'un paquet de données. Par conséquent, le choix de la taille du paquet aura un impact évident sur la surface du routeur.

Soit f la taille en bits d'un flit de données. La figure 4.20 illustre un exemple de transmission d'un paquet de 5 flits, en définissant dans le routeur esclave un buffer de taille équivalent à $5f$ bits. Un routeur est alors capable de stocker un paquet complet.

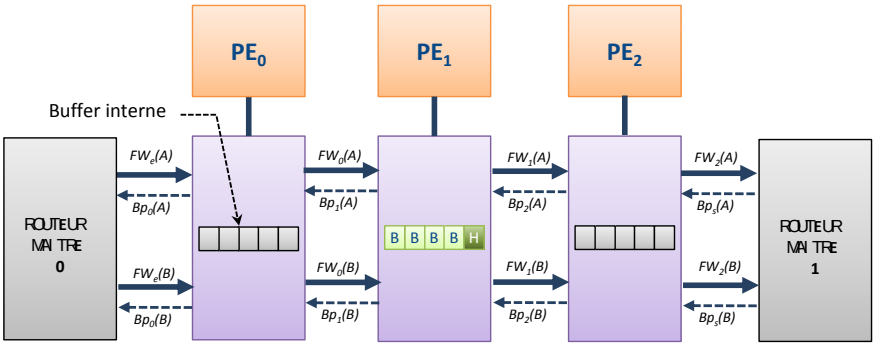


FIGURE 4.20: VCT pour un paquet de taille $5f$ bits et des buffers de taille $5f$ bits

La figure 4.21 illustre le chronogramme de la transmission du paquet.

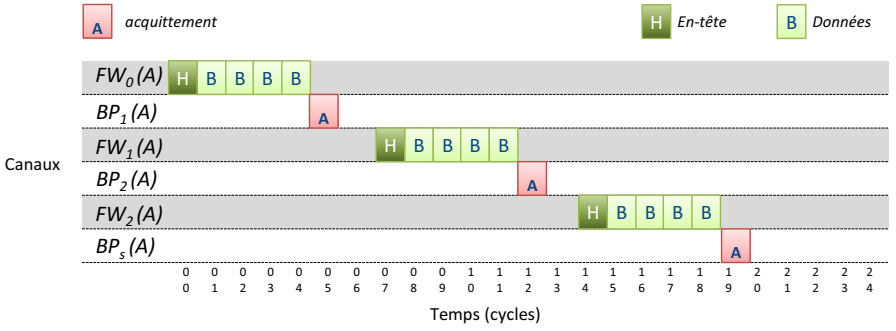


FIGURE 4.21: Transmission d'un paquet avec la méthode Store-and-Forward

Dans cette technique, chaque routeur doit attendre que le paquet soit arrivé intégralement avant de le transmettre au routeur suivant. Un signal A d'acquittement permet d'informer la réception complète du paquet. La latence de réception du paquet est alors proportionnelle à la fois à la longueur du paquet L (en nombre de flits) et à la longueur du chemin de données mesuré en terme de sauts entre les routeurs, appelés *hops*.

Avant que chaque paquet puisse être envoyé, il est nécessaire pour chaque routeur d'effectuer une allocation des ressources. Ces ressources sont principalement le canal de communication et le buffer pour stocker le paquet à réceptionner.

Notons H le nombre de *hops* requis pour faire transiter un paquet de données. Soit t_w le temps d'attente nécessaire pour effectuer les allocations nécessaires entre routeurs et t_f le temps pour transmettre le paquet sur le canal de communication. Soit b la bande passante du canal de communication, mesurée en flits par cycle.

Pour une technique SAF, la latence minimum T_0 d'un paquet pour traverser un chemin de données de longueur H est de [15] :

$$T_0 = H(t_w + t_f) = H(t_w + \frac{L}{b}) \tag{4.4}$$

Dans notre exemple, le nombre de saut H égale à 4 et le temps t_w est de 1 cycle horloge entre chaque paquet sur le chronogramme 4.21.

La technique *Virtual Cut Through* (VCT) autorise le routeur à transmettre les flits consituant un paquet avant même de recevoir le paquet dans son intégralité.

La figure 4.22 illustre un exemple de transmission du paquet avec la technique VCT.

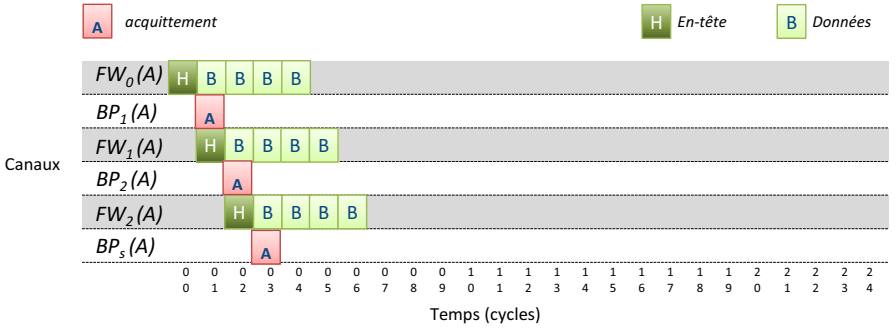


FIGURE 4.22: Transmission d'un paquet avec la méthode Virtual Cut Through

Nous observons que l'acquittement s'effectue dès le premier flit transmis. Il en résulte ainsi une latence de transmission réduite par rapport à la précédente technique SAF. Avec cette technique, la latence minimum T_0 devient [15] :

$$T_0 = H * t_w + t_f = H * t_w + \frac{L}{b} \tag{4.5}$$

Cependant, chaque acquittement implique une allocation mémoire dans le routeur égale à la taille du paquet à transmettre. La technique VCT n'apporte pas d'amélioration concernant la taille des buffers mémoires qui est au moins égale à la taille du paquet de données.

La technique *Wormhole Switching* (WS) permet de travailler avec des buffers réduits à un nombre finis de flits. Tant qu'il n'y a pas de congestion dans le routeur, cette technique se comporte de façon similaire à la technique du *Virtual Cut Through*. Un flit est ainsi automatiquement envoyé au routeur voisin dès que celui peut l'accueillir. Les allocations s'effectuent à une granularité mémoire de la taille du flit. Dans cette technique, l'en-tête contient l'information de routage et il se charge de réserver progressivement un chemin de données à chaque routeur.

La figure 4.23 illustre un exemple avec des routeurs dont le buffer mémoire est réduit à une taille ne pouvant accueillir que 3 flits.

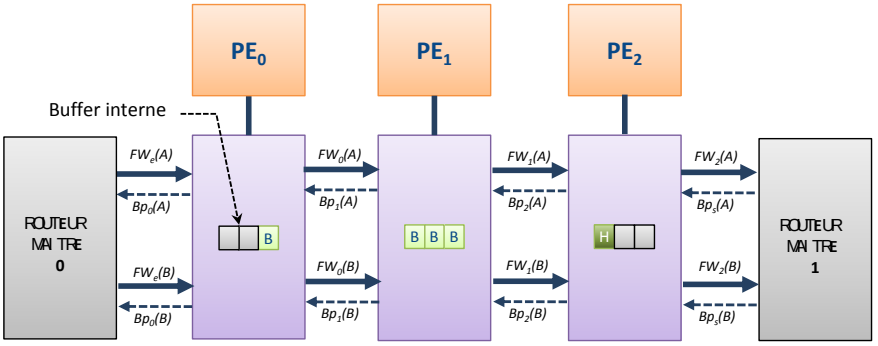


FIGURE 4.23: WS pour un paquet de taille $5f$ bits et un buffer de taille $3f$ bits

Nous observons que, contrairement à la technique VCT précédente, lors de congestion, le paquet ne va pas rester bloqué dans un seul noeud mais est étalé tout au long de son chemin de données car aucun routeur ne peut stocker complètement le paquet. Ainsi, la taille des buffers est indépendante de la taille du paquet. En pratique, l'implémentation de cette technique nécessite de mettre en oeuvre un pipeline dans chaque routeur afin de temporiser les flits entrants pendant l'allocation des ressources (mémoire, chemin de données).

Le chronogramme est ainsi similaire à celui présenté en figure 4.22 avec la même latence minimum T_0. C'est pourquoi la technique *Wormhole Switching* est la plus utilisée dans les réseaux sur puce.

Cependant, du fait de l'étalement du paquet tout au long du chemin de données, les situations de blocage (*deadlock*) apparaissent plus fréquemment.

Pour réduire les situations de blocage de la technique de *Wormhole*, une solution consiste à ajouter des canaux de transmission supplémentaires dans les routeurs et à multiplexer temporellement différents chemin de données. Il s'agit d'une technique basée sur le concept de canaux virtuels (*virtual channels*).

La figure 4.24 illustre un exemple de réseau utilisant ce principe avec 3 canaux virtuels. Deux paquets multiplexés de taille 5 flits, sont transmis au travers de la même liaison physique. Ces deux paquets sont étalés tout au long du même chemin de données physique mais sur des canaux virtuels différents.

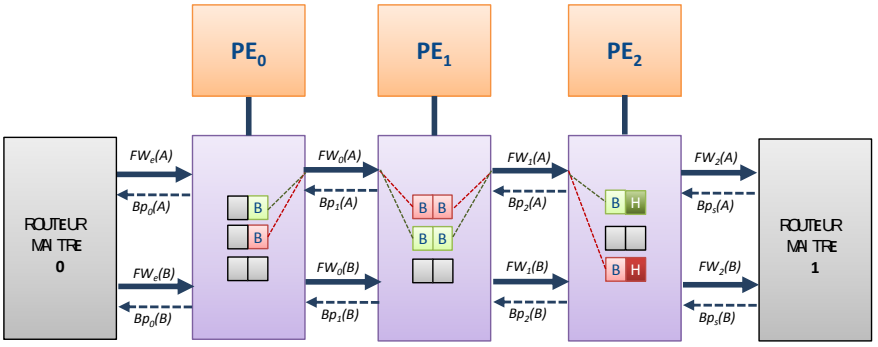


FIGURE 4.24: Exemple de deux paquets circulant avec trois canaux virtuels

Avec cette variante, on conserve toujours les avantages de la technique *Wormhole* concernant la taille de buffer réduite et la longueur de paquet indépendante de la taille des buffers. Les canaux virtuels permettent de démultiplier les possibilité de chemin de données en partageant le même lien physique. Ainsi, si un paquet est bloqué, il le sera uniquement dans le canal virtuel défini mais le liens physique sera toujours disponible pour les autres canaux virtuels. Concernant la latence minimimum T_0, elle s'exprime de la même façon que la technique précédente exceptée que la bande passante b du canal est partagée entre les canaux virtuels. Ainsi, b variera selon l'arbitrage et de l'occupation de autres canaux virtuels.

Cette technique nécessite cependant un arbitrage plus complexe et un espace de stockage plus important [14].

4.5.1.4 Utilisation des techniques dans le réseau proposé

Comme évoqué précédemment en section 4.4.2 par l'équation 4.2, les paquets de notre réseau peuvent être de taille importante, équivalents à plusieurs lignes d'une image. Ainsi, les techniques de *Store-and-Forward* ou de *Virtual-Cut Through* sont à proscrire pour les routeurs esclaves, car elles nécessitent d'allouer à chaque routeur un espace mémoire figé pour un réseau donné.

Les deux techniques d'aiguillage qui conviennent pour faire communiquer les noeuds maîtres par le biais des noeuds esclaves sont le *circuit switching* et le *wormhole switching*.

La technique *circuit switching* présente l'avantage de ne nécessiter aucune mémorisation. En effet, en réservant le chemin de données par des signaux de requête et acquittement, le paquet ne requiert pas obligatoirement un en-tête pour la réservation. Ainsi, les données peuvent être envoyés sans risque de collisions, comme illustré par le chronogramme 4.19. Cependant, elle présente une certain nombre de désavantages dans notre réseau.

Premièrement, cette technique implique un temps d'attente du signal d'acquittement pouvant être important selon la distance entre deux noeuds maîtres. Deuxièmement, un unique flit de requête de réservation du chemin de données est insuffisant pour décrire et ordonner les mode de fonctionnement de chaque routeur esclave dans le chemin de données.

La technique de *wormhole switching* est une solution qui nécessite de la mémoire dans chaque routeur esclave, mais autorise la réservation d'un chemin de données de manière progressive de routeur en routeur. Contrairement au *circuit switching*, elle ne nécessite pas de réservation préalable du chemin. La réservation est effectuée par la lecture de l'en-tête du paquet.

Du fait de la structure de nos paquet de données, la technique de *wormhole switching* est la plus appropriée mais nécessite d'être adaptée par rapport aux modes de fonctionnement du routeur esclave, comme définis dans la section 4.2.4.

4.5.2 Split-Wormhole switching

Nous proposons pour notre réseau une technique d'aiguillage spécifique au routeur esclave. Cette technique, que nous appelons *Split-Wormhole switching*, est une adaptation

de la technique *Wormhole Switching* (WS), appliquée aux modes de fonctionnement du routeur esclave présentés en section 4.2.4 : *Forward* (FWD), *Single Stream* (SSP), *Single Stream & Forward* (SSF) et *Multi Stream* (MS).

Pour la technique proposée, les routeurs esclaves se comportent de manière identique à la technique WS pour les modes *Forward*. Ainsi, le chemin de données est alloué progressivement et dynamiquement à chaque routeur esclave grâce à l'en-tête du paquet de données. Celui-ci contenant l'information du nombre de pixels dans le paquet, permettant de créditer un compteur de flits dans chaque routeur et de désallouer progressivement le chemin de données à la fin du décomptage.

La différence repose sur le fait que le routage ne s'effectue plus seulement en fonction de l'adresse du noeud destinataire. Il s'effectue également en fonction des instructions contenues dans l'en-tête qui définissent un séquencement d'opération à appliquer sur la donnée. Ainsi, le paquet peut être aiguillé ponctuellement vers une unité de calcul sur son chemin de données dans la direction du noeud maître destinataire.

Le mode *Single Stream* (SSP) constitue l'aiguillage le plus simple d'un paquet vers une unité de calcul. La figure 4.25 illustre un exemple d'utilisation de ce mode dans le routeur esclave 2.

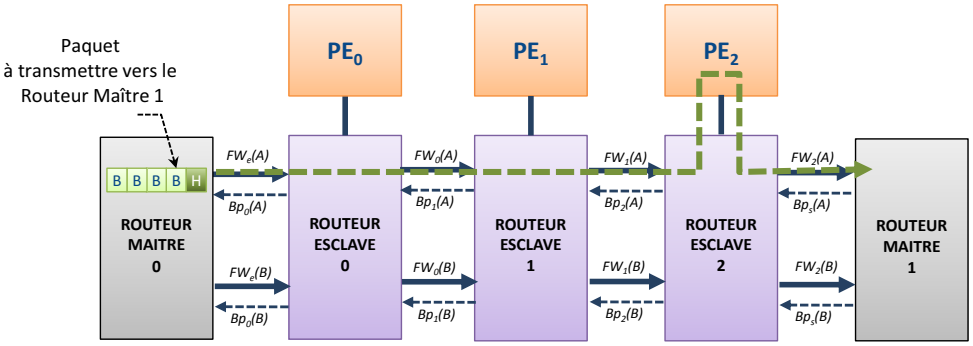


FIGURE 4.25: Aiguillage du paquet en mode SSP sur le PE2

Sur cette figure, un paquet constitué de 5 flits est émis par le routeur maître 0 vers le routeur maître 1. Les routeur maitres 0 et 1 sont séparés par trois routeurs esclaves sur lesquels sont respectivement connectés les unités de calcul PE0, PE1 et PE2. L'en-tête `[H]` du paquet, émis sur le canal $FW_e(A)$, contient une instruction indiquant le besoin d'appliquer une opération réalisable par le PE2. Ainsi, au cours de la transmission du paquet entre les deux routeurs maîtres, les deux premiers routeurs esclave (0 et 1) ne peuvent pas traiter le paquet et sont donc en mode *Forward*. Dès que le paquet est arrivé

au routeur esclave 2, celui-ci se met en mode *Single Stream* afin de traiter les données du paquet, décrites par les flits `[B]` sur la figure 4.25. Le paquet traité peut ainsi être accepté par le routeur maître 1.

Le chronogramme illustrant cet aiguillage est présenté en figure 4.26.

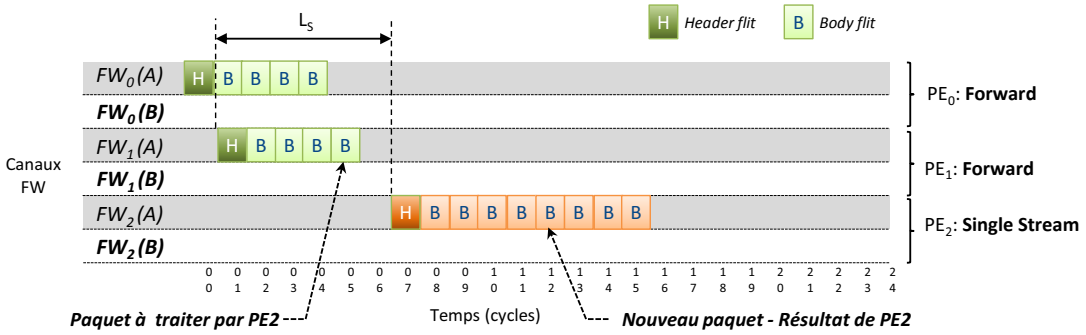


FIGURE 4.26: Chronogramme du mode SSP sur le PE2

Sur ce chronogramme, nous observons un comportement qui est similaire au *Wormhole* pour l'aiguillage du paquet en mode FWD sur les deux premiers routeurs esclaves 0 et 1. Pour cet exemple, la latence de transmission en mode FWD est illustrée à un cycle d'horloge entre les canaux $FW_0(A)$ et $FW_1(A)$ au niveau du routeur esclave 1.

Par contre, cette latence, illustrée par L_s sur la figure entre les canaux $FW_1(A)$ et $FW_2(A)$, est plus importante lors du passage dans le routeur esclave 2 en mode SSP. Cette latence s'explique par le temps de modification du chemin de données, le passage des flits dans le routeur et la latence de traitement de l'unité de calcul PE2. Nous constatons également que la taille du paquet a été modifiée par l'unité de calcul PE2 par rapport au paquet entrant. Ceci impose alors une modification de l'en-tête du paquet de données à effectuer avant son émission vers le routeur maître 1.

Par conséquent, la technique *Split-Wormhole Switching* diffère de la technique WS orginale concernant la définition de la latence minimum T_0 pour transmettre le paquet. Elle n'est donc plus seulement proportionnelle au nombre de noeuds esclaves mais est dépendante des modes de fonctionnement des routeurs esclaves spécifiés par l'en-tête du paquet de données. Soit $L_s(i)$ la latence de transmission respective de chaque routeur esclave sur le chemin de données d'un paquet. Soit H le nombre de *hops* requis pour faire transiter un paquet de données entre deux noeuds maître, L la longueur du paquet en flits et b la bande passante du canal de communication entre deux routeurs esclaves.

La latence minimum T_0 s'exprime ainsi suivant l'équation 4.6.

$$T_0 = \sum_{i=0}^{H-1} L_s(i) + \frac{L}{b} \tag{4.6}$$

Ainsi, cette latence est égale à celle de la technique WS dans le cas où tous les routeurs esclaves sont en mode *Forward* avec un temps d'allocation fixe $L_s = t_w$.

Contrairement à la technique WS, la technique d'aiguillage *Split-Wormhole Switching* n'autorise pas seulement une adaptation de chemin de donnée mono-port avec 1 port entrant et 1 port sortant. Elle permet aussi, selon le mode de fonctionnement, une adaptation de type 1 port entrant et 2 ports sortants.

C'est le cas du mode *Single Stream & Forward* (SSF) où le paquet entrant doit être dupliqué. Cette duplication est effectuée de manière à transmettre un exemplaire vers l'unité de calcul et un autre exemplaire vers le routeur esclave voisin, dans le but de traiter deux flux de données en parallèle. Deux ports de sorties doivent alors être attribués pour effectuer ce mode.

La figure 4.27 illustre un exemple d'utilisation du mode de fonctionnement SSF dans le réseau.

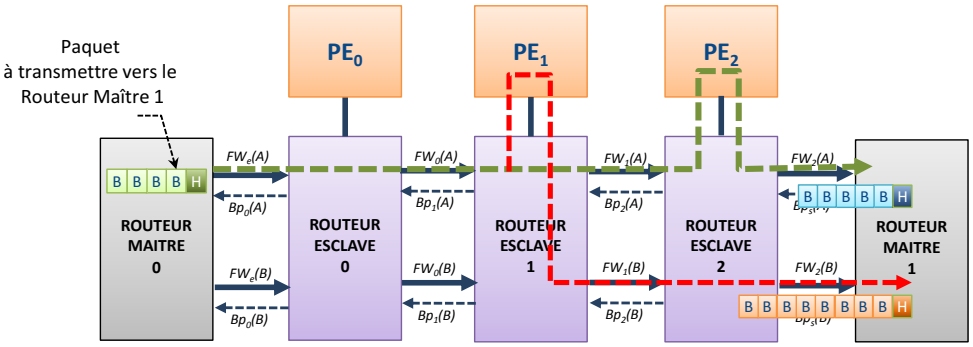


FIGURE 4.27: Ex2 : Split-wormhole switching avec deux flux en sortie

Cette figure reprend l'exemple de réseau précédent avec trois routeurs esclaves entre les noeuds maître 0 et 1. Supposons à présent que nous souhaitons traiter un paquet de données, émis par le routeur maître 0, en parallèle sur les unités PE1 et PE2 dans ce réseau. Ce calcul impose ainsi l'activation du mode SSF au niveau du routeur esclave 1. Sur la figure 4.27, nous observons ainsi le mode FWD activé pour le routeur 0 car PE0 n'est pas concerné par le calcul, et l'activation du mode SS dans le routeur esclave 2 entre les canaux $FW_1(A)$ et $FW_2(A)$.

Nous voyons que le mode SSF se distingue par la duplication du paquet de donnée entrant dans le canal $FW_0(A)$ au niveau du routeur esclave 1 vers l'unité de calcul PE1 et vers le canal $FW_1(A)$. Au final, chaque unité de calcul PE1 et PE2 fournissent deux données traitées sous forme de deux paquets différents vers le routeur maître 1.

Le chronogramme en figure 4.28 illustre l'occupation des canaux de communication pour notre exemple.

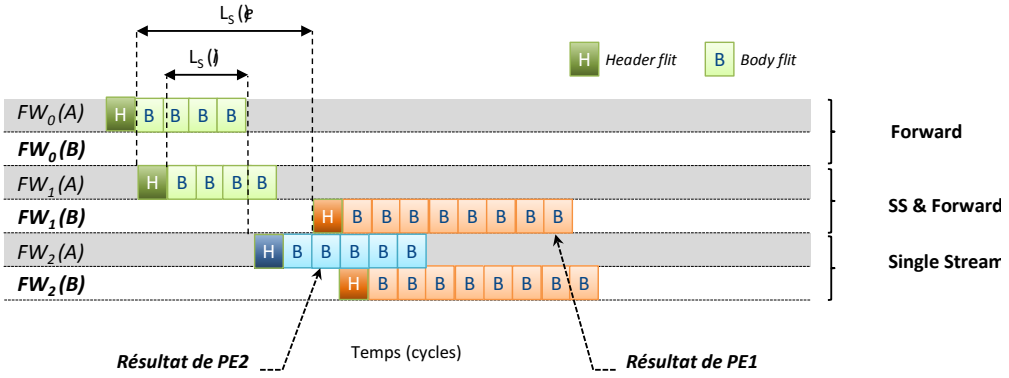


FIGURE 4.28: Chronogramme de l'exemple 2

Nous observons dans ce chronogramme 4.28, les latences de transmission du paquet traité respectives $L_s(1)$ du routeur esclave 1 entre les canaux $FW_0(A)$ et $FW_1(B)$, et $L_s(2)$ entre les canaux $FW_1(A)$ et $FW_2(A)$ du routeur esclave 2.

Dans cet exemple, nous avons illustré une différence de latence entre $L_s(1)$ et $L_s(2)$, qui s'explique par le temps d'adaptation du chemin de données ajouté à la latence propre de l'unité de calcul, qui est plus important pour le mode SSF.

La duplication du paquet de données est apparent sur le chronogramme 4.28 en observant l'occupation des deux canaux $FW_1(A)$ et $FW_1(B)$ en sortie du routeur 1. La première sortie sur $FW_1(A)$ correspond à la transmission du paquet sans traitement avec une latence réduite et la seconde sortie sur $FW_1(B)$ correspond au traitement du paquet par PE1, avec une plus grande latence de transmission apparente.

Etant donnée ces latences de transmission variables des paquets de données suivant l'aiguillage imposé au routeur, il est nécessaire de définir un algorithme de routage le plus simple et efficace possible afin de minimiser la latence d'adaptation du chemin de données. Nous décrivons ainsi dans la section suivante le routage au niveau du noeud maître et esclave de notre réseau.

4.5.3 Algorithme de routage

4.5.3.1 Principe de routage dans un réseau sur puce

Selon la topologie du réseau et le degré de liberté de chaque noeud, plusieurs solutions de chemin peuvent être empruntées par un paquet entre un noeud *maître* émetteur et récepteur. La procédure pour déterminer le chemin de données est appelée *algorithme de routage*.

Reprenons la terminologie définie au chapitre 3 précédent en section 3.3.3. Soit f_r la fonction de routage dans un réseau G tel que $G = (N, C)$ avec N représentant l'ensemble des noeuds du réseau et C l'ensemble des canaux de communication. Soit P l'ensemble des chemins de données possibles entre deux noeuds du réseau. Il correspond à une séquence finie de canaux c_i telle que $i \in \mathbb{N}$ et $\forall i, c_i \in C$.

Nous classons les routages selon deux catégories : les routages *sources* et les routages *distribués*.

Dans le cas d'un routage *source*, le chemin exact d'un paquet est préalablement calculé avant son émission dans le réseau. Le calcul du chemin de données est effectué par le noeud émetteur selon une fonction f_r définie par l'équation 4.7.

$$f_r : N^2 \rightarrow P \tag{4.7}$$

Ainsi, le paquet contient le chemin de données complet dans son en-tête. Les routeurs intervenant dans le chemin de données ne font que suivre les informations dans l'en-tête afin de transmettre les paquets entrants vers la direction imposée. Les routeurs n'ont pas de décision dans le routage.

Au contraire, dans le cas d'un routage *distribué*, le chemin du paquet n'est pas défini à l'avance mais au cours de son trajet vers le noeud de destination. La prise de décision est effectué progressivement à chaque noeud selon l'algorithme de routage défini. Ainsi, pour déterminer le canal de sortie, la fonction de routage f_r peut donc soit se baser sur le noeud courant et le noeud de destination (4.8a) ou soit se baser sur le canal de communication entrant et le noeud de destination, comme définies par les équations

4.8b. Le noeud émetteur n'a alors pas de contrôle sur le chemin emprunté par un paquet émis.

$$f_r : N^2 \rightarrow C \tag{4.8a}$$

$$f_r : C \times N \rightarrow C \tag{4.8b}$$

Qu'il soit centralisé ou distribué, un routage peut être qualifié de *statique* ou *dynamique*.

Dans le premier cas, il s'agit d'un routage déterministe sans prise en compte de l'état du réseau comme l'occupation des canaux de communication ou la distribution du trafic par exemple. Un routage statique a l'avantage d'être simple et peu coûteux en ressource (table de routage fixe en général dans chaque routeur).

A l'inverse, un routage dynamique prend en compte l'état du réseau pour sélectionner le meilleur chemin à emprunter. Les algorithmes sont ainsi plus efficaces pour améliorer la bande passante ou réduire la latence de transmission des paquets. Ce routage nécessite cependant davantage de ressources matérielles comme de la mémoire et des unités de contrôle spécifiques. Nous pouvons citer en exemple les algorithmes *odd-even* [139] ou le *fully adaptive* [140].

Dans notre modèle, nous distinguons deux niveaux de décision de routage : au niveau du noeud maître et au niveau du routeur esclave.

Pour implémenter des applications dans notre réseau, le routage au niveau des noeuds esclaves doit être distribué et dynamique. De la définition de l'en-tête de nos paquet de données, il s'agit d'un routage dynamique particulier car le chemin de données est décidé dynamiquement en fonction des instructions dans l'en-tête et du type d'unité de calcul connecté au noeud esclave. Le routage du noeud esclave est alors plus précisément *dynamique* et *semi-distribué*. En effet, le routeur maître impose un séquencement des opérations sur la donnée qui n'est réalisable qu'en fonction de la disponibilité des unités de calcul sur les routeurs esclaves. Ce séquencement est imposé par l'application sans connaissance du positionnement physique exact des unités de calcul sur les routeurs esclaves.

4.5.3.2 Routage au niveau noeud maître

Le principe de routage des flits d'un paquet de données au sein d'un noeud maître dépend de l'adressage du paquet, la disponibilité des canaux de comunication pour transmettre le paquet en sortie, et de l'*état* de traitement du paquet. Un paquet a connaissance de l'adresse de destination par l'identifiant du noeud maître destinataire. L'état de traitement du paquet est identifié grâce à son en-tête en consultant les opérations qui ont été effectuées sur la donnée, par le champ `NB PASSES` de chaque instruction, décrit en section 4.4.2.

Le pseudo-code simplifié du routage des flits pour un routeur maître est présenté dans l'algorithme 1.

Cet algorithme utilise les fonctions et procédures suivantes :

- `Init(ft)` qui initialise des registres du routeur par les valeurs `nbp`, `inst`, `atb` à partir de la valeur `ft` des flits décodées de l'en-tête
- `Transmit(n)` qui active l'adaptation du chemin pour transmettre sur le canal `n` sélectionné
- `Match(ft,M)` qui teste de la destination du paquet au routeur maître `M`
- `Store(ft)` qui permet la mémorisation des flits de valeur `ft` du paquet
- `Select_chan()` qui sélectionne un canal `n` pour la sortie du paquet
- `Load()` qui permet de charger un nouvel en-tête pour le paquet

Après une initialisation des valeurs `nbp` (nombre de pixels dans le paquet), `inst` (instructions pour le paquet), `atb` (attributs pour le paquet) décodées et fournies dans l'en-tête par la fonction `Init()` en ligne 2, le routeur maître analyse dans un premier temps si il est le destinataire du paquet de donnée avec la fonction `Match(flit, p)` en ligne 3.

Nous rappelons qu'un paquet est complètement traité lorsque le champ `NB PASSES` est à la valeur 0 pour toutes les instructions dans l'en-tête. Si nous avons ce cas, alors la variable d'état `p` de traitement du paquet est à 1.

Si le routeur maître n'est pas le destinataire, il adapte son chemin de données afin de transmettre l'intégralité du paquet en sortie vers un routeur voisin, sur un canal de communication disponible, avec la fonction `n = Select_chan()` en ligne 12. Le routeur

Algorithme 1 : Routage dans le noeud *maître*

Entrée : ft : valeur du flit en entrée ; M : valeur de l'identifiant du noeud maître recevant le flit

Sortie : nbp, inst, atb, p : valeurs décodées de l'en-tête et état de traitement du paquet

Variables :

- `nbp`: nombre de pixels dans le paquet;
- `inst`: instructions pour le paquet;
- `atb`: attributs pour le paquet;
- `p`: indicateur de traitement complet du paquet;

Fonctions :

- `Init(ft)` : initialisation des registres du routeur par les valeurs [nbp, inst, atb] décodées de l'en-tête;
- `Transmit(n)`: adaptation du chemin pour transmettre sur le canal `n` ;
- `Match(ft,M)` : test de la destination du paquet au routeur maître `M` ;
- `Store(ft)`: mémorisation du paquet;
- `Select_chan()` : sélection d'un canal `n` de sortie;
- `Load()` : chargement d'un nouvel en-tête pour le paquet;

```
1  tant que (flit entrant appartient à l'en-tête du paquet) faire
       //(1) décodage et initialisation des registres
2      [nbp, inst, atb, p ] ← Init(ft) ;
       //(2) verification de la destination du paquet et de son traitement
3      si Match(ft,M) et p =1 alors
4          si (aucun canal de sortie disponible) alors
               //enregistrement des flits en mémoire
5              Store(ft) ;
6          sinon
7              si (aucune nouvelle instruction à appliquer) alors
                   //sortie du paquet à l'extérieur
8                  Transmit(canal de sortie externe) ;
9              sinon
                   //paquet à transmettre sur le canal n
10                 n ← Select_chan() ;
11                 Transmit(n) ;
                   //édition d'un nouvel en-tête avec nouvelles instructions
12                 Load() ;
13             fin
14         fin
15     sinon
           //paquet à transmettre sur le canal n
16         n ← Select_chan() ;
17         Transmit(n) ;
18     fin
19 fin
```

modifie son chemin de donnée afin de la transmettre sur le canal `n` selectionné dans le même sens de circulation, avec la fonction `Transmit(n)` en ligne 13.

Dans le cas où le routeur maître est le destinataire (i.e. l'adresse de destination du flit contient la valeur `M` dans notre exemple), et que le paquet a été complètement traité par des PEs sur les routeurs esclaves, il y a deux possibilités.

Si aucun canal de communication en sortie n'est disponible (ligne 4) alors les flits du paquet de données sont stockés en mémoire. Dans le cas contraire, si la donnée du paquet doit encore subir de nouvelles opérations suivant l'application implémentée, le routeur doit réémettre le paquet avec un nouvel en-tête contenant de nouvelles instructions, avec la fonction `Load()` en ligne 12. Dès que toutes les opérations requises pour une application ont été appliquées, le routeur maître destinataire peut transmettre ce paquet en sortie du réseau (ligne 8).

4.5.3.3 Routage au niveau noeud esclave

Comme précisé dans l'algorithme précédent, un paquet n'est modifié par un routeur maître à condition que les données du paquet ont été complètement traitées. Si c'est le cas, le routeur maître peut alors ajouter dans l'en-tête de nouvelles opérations à appliquer. Ce paquet est traité par les unités de calcul connectées aux routeurs esclaves.

Au niveau du routeur esclave, l'adaptation du chemin de données est déduite des instructions du paquet, de la disponibilité des canaux de communication et de l'état d'occupation du PE. Cette adaptation réalise les modes de fonctionnement FWD, SSP, SSF et MS, définis en section 4.2.4 disponibles pour le routeur esclave.

Le pseudo-code simplifié du routage des flits dans un routeur *esclave* est présenté dans l'algorithme 2.

Cet algorithme 2 utilise les fonctions et procédures suivantes :

- `Init` qui initialise les registres avec les valeurs `nbp`, `op`, `atb` à partir de la valeur `ft` des flits décodées de l'en-tête
- `MatchOP(ft,op)` qui teste de la capacité du PE pour réaliser l'opération `op`
- `Detect(ft)` qui détecte le mode de fonctionnement requis pour traiter le paquet
- `Select_chan()` qui sélectionne un canal de sortie `n`

– `Transmit(n)` qui adapte le chemin pour transmettre le paquet sur le canal `n`

– `Adapt_datapath(mode)` qui active le mode de fontionnement `mode` du routeur esclave

– `Edit_header(ft)` qui modifie les flits de l'en-tête pour valider l'opération sur le paquet

Algorithme 2 : Routage dans le noeud *Esclave*

Entrée : ft : valeur du flit en entrée : busy : occupation du PE
Sortie : nbp, atb, op, n, mode
Variables :
- busy: occupation du PE;
- nbp: nombre de pixels dans le paquet;
- atb: attributs pour le paquet;
- op: opération à appliquer;
- n: canal selectionné;
- mode: mode de fonctionnement FWD, SSP, SSF, MS;

Fonctions :
- Init: initialisation des infos du paquet;
- MatchOP(ft,op) : test de la capacité du PE pour réaliser l'opération op ;
- Detect(ft) : détection du mode de fonctionnement du paquet;
- Select_chan() : sélection d'un canal de sortie;
- Transmit(n) : adapte le chemin pour transmettre sur le canal n ;
- Adapt_datapath(mode) : active le mode de fonctionnement mode ;
- Edit_header(ft) : modification de l'en-tête pour valider l'opération sur le paquet;

```
1  tant que (flit entrant appartient à l'en-tête du paquet) faire
       //décodage, initialisation et détection du mode de fonctionnement
2      [nbp, atb, op ] ← Init(ft) ;
3      mode ← Detect(ft) ;
4      si (MatchOP(ft,op) et busy = 0) alors
           //Edition de l'en-tête et Adaptation en mode SSP, SSF ou MS
5          Edit_header(ft) ;
6          Adapt_datapath(mode) ;
7      sinon
           //Adaptation en mode Forward (FWD)
8          n ← Select_chan() ;
9          Transmit(n) ;
10     fin
11 fin
```

Après une initialisation des registres et détection du mode de fonctionnement nécessaire pour le paquet de donnée entrant, avec les fonctions `Init(ft)` et `Detect(ft)` en ligne 2 et 3, le routeur vérifie que le PE connecté est disponible (i.e. `busy=0` en ligne 4) et capable de réaliser une opération requise sur la donnée, avec la fonction `MatchOP(ft,op)`.

Si ces conditions sont remplies, le routeur édite l'en-tête du paquet de données afin de marquer les nouveaux attributs du paquet, avec la fonction `Edit_header(ft)` en ligne 5, puis adapte son chemin de données interne, avec la fonction `Adapt_datapath(mode)` en ligne 6, pour réaliser les modes de fonctionnement *Single Stream* (SSP), *Single Stream & Forward* (SSF) et *Multi Stream* (MS). Dans le cas contraire, le routeur doit se mettre en mode *Forward* (FWD) et détermine un canal de sortie pour accéder à un routeur voisin, avec la fonction `Select_chan()` en ligne 8. Il transmet ensuite le paquet afin de trouver un PE capable de le traiter, avec la fonction `Transmit(n)` en ligne 9.

4.5.4 Contrôle du flot de données dans le réseau

Le contrôle de flot de données fixe les méthodes de communication entre les routeurs du réseau. Certaines méthodes autorisent ou non la perte de données durant les transmissions. La perte de données impose la réémission des paquets du noeud émetteur si nécessaire. Nous choisissons une méthode *sans perte* dans le but d'optimiser la bande passante pour le traitement des paquets par les unités de calculs sur les noeuds esclaves. Dans ce but, la gestion des buffers dans les routeurs est importante. Elle doit s'effectuer de manière à n'avoir aucune perte de flits pouvant être causée par une incapacité de stockage dans un routeur. Un signal, appelé *backpressure*, est typiquement utilisé pour prévenir un noeud émetteur de l'incapacité d'un noeud récepteur à accueillir une donnée. Dans notre architecture, il s'agit des signaux Bp_x sur la figure 4.3 entre les routeurs.

Il existe différentes méthodes pour contrôler le flot de données dont les plus connues utilisent une technique basée sur le *crédit*, une technique de type *marche-arrêt* ou une technique d'*acquittement* [130].

La technique basée sur un crédit consiste à associer à chaque noeud un compteur de flits. Avant chaque émission, ce compteur est crédité par le noeud récepteur voisin. Au cours de l'émission des flits, ce compteur est décrémenté et lorsqu'il atteint le seuil de zéro, il arrête l'envoi de flits. Le compteur du noeud émetteur n'est seulement re-crédité qu'à chaque flit reçu correctement par le noeud récepteur.

La technique basée sur acquittement consiste à émettre un flit et attendre un hypothétique signal d'acquittement (ack) du noeud récepteur. Si celui-ci ne peut pas le recevoir, alors il renvoie un signal de refus (nack) et l'émetteur doit réémettre la même donnée.

Nous pouvons citer l'exemple de réseau proposé dans [141] ou l'exemple du protocole *go-back-n* utilisé dans [142].

La technique basée sur un signal marche/arrêt est la solution la plus économique en ressource pour communiquer l'état des buffers d'un noeud récepteur vers un noeud émetteur. Elle consiste à n'utiliser qu'un seul signal par le noeud récepteur pour informer le noeud émetteur de la possibilité d'émission. Il s'agit de la solution la plus adaptée à notre réseau. Cependant, la génération de ce signal ne s'effectue pas simplement que sur l'état du buffer mais aussi sur son mode de fonctionnement dans le cas du routeur esclave.

Un paquet de taille erronée pourrait bloquer le réseau du fait de la mauvaise correspondance avec le nombre de pixels spécifiés dans l'en-tête. Pour améliorer la robustesse de notre réseau, nous introduisons une règle supplémentaire de délai d'attente (*timeout*) entre les flits ce qui permet de libérer les routeurs dans le cas d'émission de paquets erronés. Nous réduisons ainsi le blocage des routeurs qui peuvent rester en attente permanente de flits.

4.6 Conclusion

Dans ce chapitre, nous avons proposé un nouveau modèle d'architecture de réseau de communication basé sur deux types de routeurs : le routeur maître et le routeur esclave. Les noeuds maître sont connectés de façon indirecte par des noeuds esclaves. Dans ce réseau, les échanges principaux de paquets s'effectuent entre les noeuds maîtres et le traitement de ces paquets s'effectuent dans les noeuds esclaves contenant des unités de calcul. Il s'agit d'un réseau de communication capable de supporter plusieurs flux en parallèle. En particulier, le routeur esclave est capable de modifier dynamiquement son chemin de données afin de pouvoir traiter un paquet avec des unités de calculs pipelinées ou en parallèle.

Nous avons défini une structure de paquet afin de pouvoir transmettre des images (ou partie d'image) entre les noeuds principaux maîtres. L'en-tête du paquet permet l'identification du paquet de données sur le type de source d'image, le positionnement temporel de l'image associée et la dernière opération appliquée.

Une nouvelle méthode de communication des commandes sur les routeurs esclaves a été proposée en associant directement des instructions avec la donnée concernée. Ces instructions correspondent à un séquencement d'opération à appliquer sur la donnée en fonction de l'application à implémenter.

A partir de cette structure de paquet, nous proposons une nouvelle méthode d'aiguillage, appelée *Split-Wormhole switching*, adaptée à notre routeur esclave. L'algorithme de routage dynamique associée à ce routeur permet de décider de la modification du chemin en fonction de l'adresse de destination et des instructions contenues dans l'en-tête.

La conception de routeur disposant d'un routage dynamique est coûteuse en terme de ressource logique et mémoire. De plus, étant donné que la prise de décision s'effectue à chaque noeud du réseau, cette technique introduit une latence qu'il est nécessaire de minimiser. Le routeur esclave étant la composante la plus utilisée dans notre modèle, il est donc important de définir une architecture à la fois performante en temps et avec une consommation en surface réduite.

Le chapitre suivant se consacrera alors à l'étude de ce routeur dans le but de proposer une architecture implémentable et évaluée en terme de surface et temps.

Chapitre 5

Conception d'un routeur multi-flux dynamiquement adaptable

5.1 Introduction

Ce chapitre est consacré à la conception de la micro-architecture du routeur esclave. Le routeur esclave est l'unité la plus utilisée dans le réseau et permet la connexion de différentes unités de calcul capables de traiter les données transitant entre deux routeurs maîtres. Sa micro-architecture doit ainsi être soignée de manière à garantir les performances en temps de transmission des paquets et d'adaptation du chemin, avec une surface minimale.

Dans ce chapitre, nous discutons, dans une première partie, des méthodes d'adaptation d'un routeur de données dans le contexte des NoCs. Dans une seconde partie, nous proposons une définition d'un routeur multi-flux dynamiquement adaptable suivie de la description d'un modèle architectural respectant les contraintes de conception du routeur esclave, définies au chapitre précédent pour notre réseau. Nous présentons, par la suite, une proposition d'implémentation de ce routeur avec une comparaison de cette solution par rapport à une conception plus fréquente dans la littérature. Nous terminons ce chapitre par une évaluation de l'architecture en termes de temps d'adaptation et de surface dans le cadre d'une implémentation matérielle sur FPGA.

5.2 Routeur adaptable dans un NoC

D'une manière générale, un routeur est conçue pour rediriger des paquets de données provenant d'un port en entrée vers un de ses ports en sortie. Dans un contexte d'implémentation sur puce, comme illustré en figure 5.1, la micro-architecture interne du routeur repose principalement sur des unités de commutation (*switch*), des unités de mémorisation avec des registres en entrée-sortie et des unités de contrôle, afin de calculer le chemin de données et allouer les ressources nécessaires [14]. Une unité de commutation (switch) peut être implémentée par un ensemble de multiplexeurs définissant un nombre fini de chemins de données réalisables au sein d'un routeur.

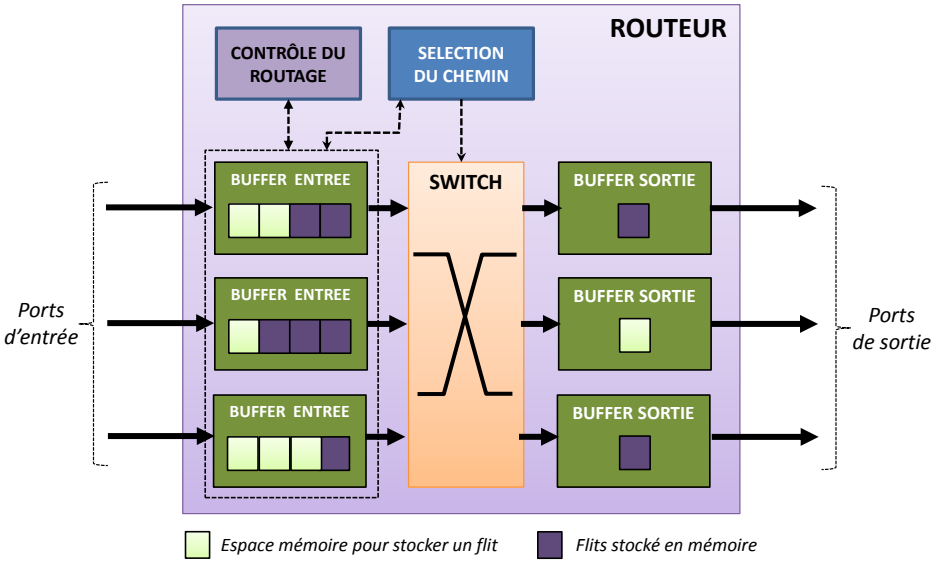


FIGURE 5.1: Structure d'un routeur sur puce

La structure d'un routeur est généralement pipelinée et on classe ses unités internes en deux sous-groupes : un groupe dédié au *chemin de donnée* correspondant aux ressources associées (i.e. les buffer d'entrée et de sortie et switch) et un groupe dédié au *contrôle* contenant les modules qui coordonnent les mouvements des paquets (i.e. unités de sélection du chemin et de contrôle du routage).

La conception de son architecture est naturellement dépendante de la technique d'aiguillage choisie pour le réseau, des protocoles de communication entre les routeurs, du nombre de ports d'entrée-sortie, et de l'algorithme de routage utilisé dans le réseau de communication. Ces paramètres impactent la surface de silicium nécessaire, avec une influence sur l'espace de mémorisation nécessaire et sur la taille du switch.

Dans le contexte des NoCs, nous distinguons l'adaptabilité *pré-synthèse* et *post-synthèse* de l'architecture d'un routeur sur puce.

L'adaptabilité *pré-synthèse* caractérise les différents paramètres qui peuvent être définis avant la synthèse du routeur. Ces paramètres peuvent définir le nombre de ports ou la taille des bus par exemple, mais aussi la méthode de contrôle dans la structure interne du routeur suivant le type de réseau choisi [114]. En se basant sur ce paramètrage, cette adaptabilité peut fixer les premières limitations du réseau sur puce en terme de topologie autorisée ou des algorithmes de routage pouvant être mis en oeuvre.

L'adaptabilité *post-synthèse* caractérise un routeur par sa capacité architecturale à répondre dynamiquement à son environnement. Cette réponse peut être sollicitée par une commande externe au routeur ou peut être réalisée de manière autonome. Dans ce dernier cas, nous parlons plus précisément de routeur *auto-adaptable* comme défini dans [120].

Dans la majorité des réseaux sur puce, cette auto-adaptation qualifie généralement un routeur capable de modifier son fonctionnement selon un algorithme de routage adaptatif. Il est ainsi capable de modifier le routage des paquets en fonction de l'état du réseau dans le but d'améliorer la bande passante par exemple. Un autre exemple consiste à mettre en oeuvre un réseau de communication qui est tolérants aux fautes [120] et capable de s'adapter suivant des modifications dynamiques des unités de calcul (PEs) à chaque noeud du réseau. La modification dynamique d'une unité de calcul peut se faire par reprogrammation [15, 124, 143] ou par reconfiguration dynamique [112, 144], selon la technologie définie dans l'architecture de calcul.

Dans le cas d'un réseau auto-adaptable, les routeurs ont des informations sur l'état du réseau comme la position des goulots d'étranglement de données ou d'un routeur non fonctionnel et défecteux. L'adaptation est alors réalisée en fonction de l'état du réseau et de l'adresse de destination. Selon l'algorithme de routage, le routeur sélectionne le port de sortie permettant d'optimiser le chemin de donnée du paquet en fonction de l'état du réseau.

Dans le cadre de notre réseau, les routeurs esclaves disposent d'une adaptabilité pré-synthèse et d'une adaptabilité post-synthèse. L'adaptabilité pré-synthèse est nécessaire

pour définir la taille des bus de données et le nombre de ports de communication selon le nombre de sources d'image et la bande passante requise entre les routeurs.

L'adaptabilité post-synthèse de ce routeur est caractérisée par sa capacité à auto-adapter ses chemins de données internes entre les ports de communication. Un routage dynamique défini lui permet ainsi de modifier les chemins, en fonction des instructions contenues dans l'en-tête d'un paquet de données entrant, de la disponibilité de l'unité de calcul et de celle des ports de communications en sortie.

L'utilisation d'un algorithme de routage adaptatif nécessite toutefois une logique de traitement plus avancée pour le décodage et le contrôle par rapport au choix d'un algorithme de routage déterministe. De plus, chaque décision de routage est calculée dynamiquement à chaque noeud du réseau et elle implique une latence supplémentaire cumulée en fonction de la longueur du chemin de données. Dans le contexte où il est nécessaire de faire transiter des flux de données en parallèle pour des contraintes applicatives, il est par conséquent nécessaire de définir un choix de structure de routeur capable de répondre aux contraintes de performance en temps, avec une surface minimale.

5.3 Proposition architecturale d'un routeur multi-flux dynamiquement adaptable

5.3.1 Définition d'un routeur multi flux dynamiquement adaptable

Soit Z l'architecture de calcul sur puce définie pour réaliser un ensemble $\mathbb{A}$ fini d'applications de traitement d'image. Ces applications permettent la visualisation d'une à k sources d'images issues de k capteurs.

Pour implémenter ces applications, l'architecture Z dispose de plusieurs unités de calcul PE_i qui se communiquent sur un réseau G. Les unités de calcul sont respectivement connectées au réseau G par l'intermédiaire de routeurs esclaves R_i. Le réseau G est défini de manière à supporter k flux de données pouvant être traités en parallèle, et impose que chaque PE_i soit définie avec m données en entrée pour une seule donnée en sortie.

Une application $A_n \in \mathbb{A}$ (n $\in \mathbb{N}$) est décrite sous forme d'un ordonnancement d'opérations avec la condition que chaque opération soit réalisable par une ou plusieurs unités de calcul PE_i de l'architecture Z. Le routeur R_i doit ainsi être capable de modifier son chemin de données interne pour utiliser l'unité de calcul suivant l'application à réaliser pour un instant donné.

Nous définissons un routeur multi flux dynamiquement adaptable comme un routeur capable :

1. de gérer k flux de données entrants en parallèle
2. de connecter une unité de calcul PE_i avec m ports en entrée tel que $m \leq k$ et un port de sortie
3. d'adapter le chemin de données de k flux de données entrants en fonction de la donnée, de l'état d'occupation de PE_i et de la disponibilité d'un port de sortie

De ce fait, nous qualifions le routeur esclave R_i comme un routeur multi-flux et dynamiquement adaptable. La figure 5.2 illustre schématiquement un exemple de routeur esclave R_i.

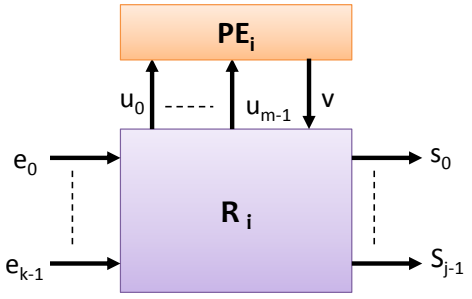


FIGURE 5.2: Schéma des entrées et sorties du routeur R_i

Ce routeur est capable d'accepter k données en entrée $\{e_0, ..., e_{k-1}\}$ et une donnée v provenant de l'unité de calcul. Il dispose de j ports de sortie vers le réseau et de m ports de sorties vers le PE tels que $j \leq k$ et $m \leq k$. Le routeur comporte ainsi au total $k+1$ ports d'entrée et $j+m$ ports de sortie.

5.3.2 Proposition architecturale du routeur esclave

Pour le réseau proposé dans ce manuscrit, nous rappelons que les données sont transmises sous forme de paquets par les noeuds maîtres vers les routeurs esclaves. Ces routeurs

esclaves sont de type multi-flux dynamiquement adaptable. Nous nous concentrons dans ce chapitre sur la conception du routeur esclave. Celui-ci est associé à une unité de calcul définie pour effectuer un ou plusieurs types d'opérations. Suivant l'application à réaliser, il doit être capable d'adapter son chemin de donnée interne afin de réaliser un séquencement correct des opérations.

Pour minimiser les risques de blocage d'un flux de données, nous avons imposé une règle de construction du réseau en section 4.2.5, de manière à appairer un port d'entrée et un port de sortie. Ainsi, dans le schéma de routeur R_i en figure 5.2 précédente, le paramètre j doit être égal à k pour la conception du routeur esclave. Pour chaque port k_i, nous associons alors les ports u_i et j_i respectivement vers le PE et vers un routeur voisin.

Une vue globale de l'architecture proposée pour le routeur esclave est présentée dans la figure 5.3.

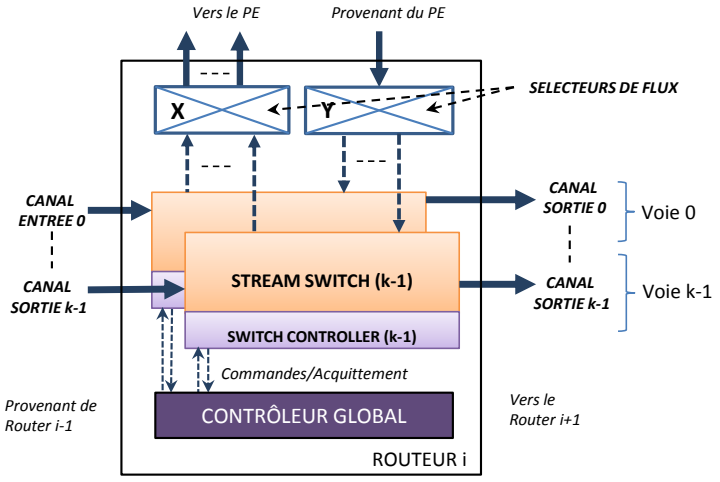


FIGURE 5.3: Architecture du routeur esclave i avec k voies de communication

Nous appelons *voie* de communication, le chemin de données reliant chaque couple défini de ports d'entrée-sortie (e_i,s_i). Ainsi, un routeur esclave est capable de s'adapter pour réaliser k voies indépendantes pouvant véhiculer k flux de données en parallèle. Au total, cette architecture comporte $k+1$ ports de communication entrants et $k+m$ ports de communication sortants.

L'architecture proposée est constituée de trois types d'unités :

– des unités de *commutation de flux* (*Stream Switch*) pour chaque voie
– des *sélecteurs de flux* (X et Y) en entrée et sortie du PE
– un *contrôleur* global

Les unités de commutation de flux, appelées *Stream Switch* sur la figure 5.3, sont associées à chaque voie de communication. Elles ont pour rôle de décoder les paquets de données entrants et rediriger les flux de données vers le PE et/ou le port de sortie associé. Ces unités fonctionnent de manière indépendantes avec un contrôleur local, appelé *Switch Controller*, qui s'occupe de gérer l'adaptation du chemin de données pour la voie de communication.

Les règles de construction du réseau, en section 4.2.5, imposent une unidirectionnalité des ports de communication. Cette unidirectionnalité permet de maintenir une simplicité dans la structure du routeur et ainsi optimiser les performances en fréquence de fonctionnement. En effet, par rapport à une voie, un paquet entrant ne possède que deux degrés de liberté : un degré vers l'unité de calcul et un autre vers le port de sortie correspondant à la voie.

L'ensemble des unités du routeur est supervisé par un contrôleur global qui s'occupe principalement d'arbitrer les accès sur les ports de sortie et de commander les unités de sélection de flux. Le contrôle du routeur esclave est ainsi hiérarchique avec une decision *locale* de routage au niveau de chaque voie et une décision de routage *globale* permettant d'arbitrer si besoin les adaptations requises pour chaque voie.

Les unités de sélection de données en entrée et en sortie du PE sont représentées par les sélecteurs de flux X et Y sur la figure 5.3. Leur rôle est d'aiguiller correctement les flux en entrée et en sortie du PE suivant des commandes provenant des *Stream Switch* et du contrôleur global.

Nous pouvons remarquer que seule la sortie de l'unité de calcul possède un degré de liberté maximum sur toutes les sorties vers le réseau, dans le but de réaliser tous les modes de fonctionnement (FWD, SSP, SSF et MS) spécifiés pour le routeur esclave.

Il par conséquent nécessaire de définir des règles d'arbitrage pour gérer les autorisations d'accès sur une voie de communication disponible afin d'éviter toute collision de données. Cet arbitrage est nécessaire pour des adaptation *multi-voies* comme le cas du mode SSF (*Single Stream & Forward*) présenté au chapitre 4 précédent.

5.3.3 Contrôle et arbitrage des accès

5.3.3.1 Evolution de l'état du contrôleur global

Nous avons décrit dans le chapitre 3 précédent, les quatres modes de fonctionnement du routeur esclave : *Forward* (FWD) , *Single Stream* (SSP) , *Single Stream & Forward* (SSF) et *Multi-Stream* (MS).

Le contrôleur global reçoit des requêtes d'accès à l'unité de calcul provenant des contrôleurs locaux de chaque voie de communication. Ces requêtes proviennent du décodage des paquets de données entrants dans chaque voie. Une prise de décision globale est donc requise pour tous les modes de fonctionnement excepté le mode *Forward* (FWD). En effet, celui-ci correspond à un mode où le flux de données ne fait que traverser le routeur sans aucun traitement par l'unité de calcul.

Le contrôleur global évolue selon quatre états principaux de fonctionnement :

1. état attente
2. état attente MS
3. état séquentiel
4. état parallèle

L'état *attente* correspond à un état d'attente de requêtes d'accès au PE connecté au routeur. L'état *attente MS* est un état d'attente particulier au mode de fonctionnement MS. Dans cet état, le contrôleur attend que toutes les données à traiter en parallèle sont arrivées dans le routeur avant de pouvoir lancer le calcul dans le PE.

Selon le mode de fonctionnement activé et le nombre de voies utilisé, nous distinguons deux types d'état : *séquentiel* et *parallèle*. L'état *séquentiel* correspond à une opération n'impliquant qu'une seule voie de communication. Il correspond alors uniquement au mode SSP. L'état *parallèle* correspond à un fonctionnement multi-voie, adaptant le chemin de données de plusieurs voies, pour faire transiter différents flux de données en parallèle. Cet état est activé pour les modes SSF et MS.

Le pseudo-code de changement d'état à partir de l'état d'attente, est donné par l'algorithme 3.

Algorithme 3 : Etats du contrôleur global

Entrée : fmode, nb_streams : modes demandés et nombre de flux à traiter
Sortie : state : etats du contrôleur global

Variables :
- fmode: mode de fonctionnement 02=SSP, 03=SSF, 04=MS ;
- nb_streams: nombre de flux à traiter par le PE ;
- state: état ATTENTE, ATTENTE MS, SEQUENTIEL ou PARALLELE ;

```
1  si state = ATTENTE alors
2  |  si Aucune voie disponible alors
3  |  |  state = ATTENTE ;
4  |  sinon si fmode = 02 alors
5  |  |  state = SEQUENTIEL ;
6  |  sinon si fmode = 03 alors
7  |  |  state = PARALLELE ;
8  |  sinon si fmode = 04 alors
9  |  |  si nb_streams atteint alors
10 |  |  |  state = PARALLELE ;
11 |  |  sinon
12 |  |  |  state = ATTENTE MS ;
13 |  |  fin
14 |  fin
15 fin
```

Nous remarquons que dans l'état *séquentiel*, l'en-tête est édité directement par l'unité de contrôle local associé à chaque voie de communication. Dans l'état *parallèle*, le contrôleur global choisit le port de sortie et se charge de générer le nouvel en-tête pour les données en sortie du PE.

5.3.3.2 Arbitrage des accès

Dans le cas de requêtes simultanées dans le contrôleur global, une règle de priorité est appliquée entre les voies de communication. Cette règle est basée sur un indice numérique pré-établi des voies de communication qui est défini arbitrairement avant synthèse. Par exemple, la priorité peut ainsi être accordée selon un ordre croissant des indices attribués. Cette priorité est exclusive par paquet de données afin d'éviter toute rupture du flot de flits de données avec des risques de perte de données. Ainsi, dès qu'une voie de communication dispose d'un accès au PE, elle la conserve jusqu'à la fin du traitement

complet du paquet de données entrant. Nous remarquons qu'il n'y a aucune priorité en fonction du mode de fonctionnement demandé.

Le tableau 5.1 résume la sélection de voie selon les modes de fonctionnement FWD, SSP, SSF et MS du routeur esclave. La première et deuxième colonne indiquent respectivement le nombre de voie impliqué en entrée et en sortie selon le mode activé. La dernière colonne indique le choix du port de sortie en fonction du mode.

Mode	Voie(s) Entrée	Voie(s) Sortie	
	Nombre	Nombre	Sélection du port
FWD	1 voie	1 voie	voie identique
SSP	1 voie	1 voie	voie identique
SSF	1 voie	2 voies	indice croissant
MS	n voies	1 voie	dernière voie utilisée

TABLE 5.1: Sélection des voies selon les modes du routeur esclave

D'après ce tableau, pour le mode *Forward* (FWD) et *Single Stream* (SSP), le port de sortie doit appartenir à la même voie que celui du port d'entrée, selon l'appairage défini entre les ports d'entrée et de sortie. Ainsi, pour le mode SSP, les données du paquet entrant dans une voie k sont dirigées vers le PE et les données sortantes du PE sont dirigées obligatoirement vers la sortie de cette même voie k.

Dans le cas du mode *Single Stream & Forward* (SSF), deux ports de sortie doivent être selectionnées. Pour le premier port, les données du paquet entrant sur une voie k sont envoyées à la fois vers le PE et vers le port de sortie de cette même voie k. Le second port de sortie est sélectionné en fonction de l'indice numérique de la voie et de sa disponibilité.

Enfin, dans le mode *Multi-Stream* (MS), n voies en entrée peuvent être impliquées. Cependant, les données en sortie du PE sont dirigées vers le port de sortie appartenant à la dernière voie utilisée pour transmettre la donnée en entrée à traiter en parallèle par le PE. Ainsi, en prenant l'exemple avec deux flux entrants ($n = 2$), les données du paquet entrant sur la voie k sont envoyées vers la première entrée du PE et le contrôleur global se met en attente d'un autre paquet sur une autre voie. Si la seconde donnée souhaitée arrive sur la voie j alors les données calculées par le PE sont dirigées vers le port de sortie appartenant à cette même voie j.

5.4 Traitement d'un paquet de données

5.4.1 Etapes de traitement d'un paquet de données

Comme décrit au chapitre précédent, chaque unité de commutation de flux (*Stream Switch*) doit être capable d'adapter son propre chemin de données de façon locale en fonction de l'analyse de l'en-tête d'un paquet de données entrant.

Le traitement d'un paquet de données entrant se déroule en quatre phases dont le séquencement est illustré par la figure 5.4 :

1. L'analyse de l'en-tête du paquet de données
2. L'adaptation du chemin de donnée local et global du routeur
3. La mise à jour et/ou la génération d'un nouvel en-tête
4. La transmission et le traitement des données du paquet

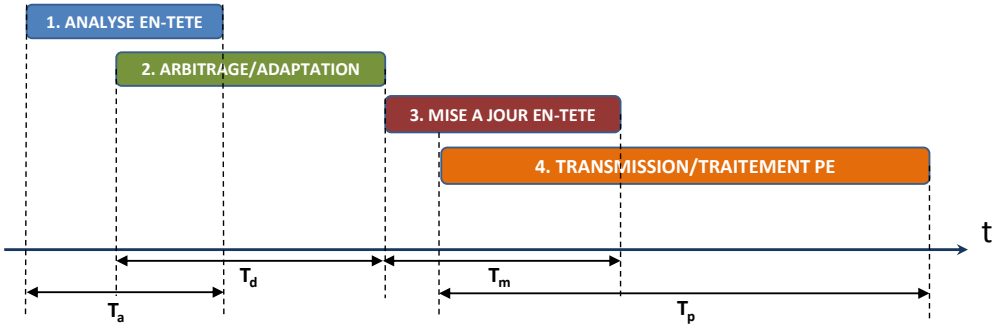


FIGURE 5.4: Etapes de traitement d'un paquet de données entrant

Les temps associés à chaque phase varient suivant l'état de fonctionnement du routeur, l'occupation des voies de communication et de l'utilisation de l'unité de calcul.

La première étape de temps T_a analyse l'en-tête complète du paquet de donnée afin d'extraire les *instructions* et les *attributs*. Les données du paquet sont pris en charge par le PE en fonction de cette analyse. Ainsi, si aucune opération décodée dans les instructions ne correspond au PE, alors le paquet de données est transmis en sortie du routeur sans aucune modification. Lorsque le PE est déjà en cours d'utilisation, le paquet est transmis directement sans besoin d'analyse et le temps T_a associé à cette étape doit être minimal.

La seconde étape de temps T_d est l'étape d'adaptation du chemin de données. Cette adaptation peut être uniquement locale à une voie de communication ou globale suivant les instructions dans l'en-tête. Dans le cas d'une demande d'adaptation globale, un processus d'arbitrage est requis entre les voies de communication. Comme mentionné précédemment, le chemin de données choisi est verrouillé entre la voie associée à l'unité de commutation et le PE connecté jusqu'à la fin du traitement du paquet. En conséquence, les autres paquets ne sont pas en attente et sont transmis directement en sortie du routeur afin de trouver un autre PE capable de les prendre en charge.

Après avoir adapté le chemin de données du routeur, l'en-tête du paquet doit être édité pour les nouvelles données à traiter. C'est l'objet de la troisième étape. Cette mise à jour permet de notifier aux autres routeurs que le traitement a bien été effectué par le PE connecté. Cette mise à jour consiste concrètement à modifier le champ correspondant au nombre d'itérations de l'instruction associée à une opération. Elle peut être plus ou moins importante sur l'en-tête selon le mode de fonctionnement requis pour le routeur. Dans le cas de besoin de transmission en parallèle de deux paquets (le paquet de donnée entrant et le paquet de donnée traité), deux en-têtes différentes doivent être générées à partie de l'en-tête du paquet entrant. Il s'agit d'une phase importante car elle assure la cohérence du séquencement des opérations sur la donnée. Cette procédure est détaillée dans la section suivante.

Cette troisième étape peut se réaliser en parallèle de la quatrième étape qui consiste à terminer la transmission complète du paquet de données et attendre la fin du traitement de l'ensemble des flits de données par le PE.

5.4.2 Mise à jour du contenu de l'en-tête

Comme indiqué précédemment, la modification de l'en-tête assure la cohérence du séquencement des opérations sur un paquet de donnée entrant. La mise à jour de l'en-tête d'un paquet repose essentiellement sur la modification de la valeur du champ contenant le nombre d'itérations associé à chaque opération. Le principe général est résumé par l'algorithme 4.

La modification de l'en-tête est obligatoire pour les trois modes de fonctionnement du routeur SSP, SSF et MS.

Algorithme 4 : Edition d'en-tête(s) dans le routeur

Entrée :
`En-tête entrant décodé dans la voie x ;`
- NBIN `[x,j]` : champs `NB PASSES` de l'instruction j du paquet entrant sur voie x ;
- TAG `[x,j]` : identificateur de parallélisme pour l'instruction de rang j du paquet entrant sur voie x ;

Sortie :
`En-têtes sortants;`
- NBOUT `[x,j]`: champs `NB PASSES` de l'instruction j du paquet sortant sur voie x ;

Variables :
- fmode: mode sélectionné (01=FWD, 02=SSP, 03=SSF, 04=MS);
- kmax: nombre d'instructions maximum par en-tête;
- op_id: identificateur de l'opération réalisée par le PE;
- PAR: indicateur de l'exécution de l'opération en parallèle avec une autre;

Fonctions :
- `Select_free_chan()` : sélection d'un canal de sortie;
- `Wait_last_chan()` : attente du paquet à traiter en parallèle par le PE;
- `Clear_instructions(z)` : toutes les instructions du paquet sur la voie z ont été traités;
- `Edit_opcode(z,op_id,0)` : remplacement de l'opération de l'instruction 0 du paquet sur la voie z par op_id ;

```
1  si fmode =02 alors
2  |  NBOUT [x,j ] ← NBIN [x,j ]-1 ;                    //mode Single Stream (SSP)
3  sinon si fmode =03 alors
4  |  NBOUT [x,j ] ← 0 ;                                 //mode SSF : 1er en-tête
5  |  y =Select_free_chan() ;                           //sélection voie sortie PE
6  |  NBOUT [y,j ] ← NBOUT [y,j ]-1 ;                    //mode SSF : 2e en-tête
7  |  i ← j +1;
8  |  tant que TAG [x,i ] = not(PAR) and i ≤ kmax faire
9  |  |  NBOUT [x,i ] ← 0 ; //annulation de toutes les opérations en parallèle
10 |  |  i ++;
11 |  fin
12 sinon si fmode = 04 alors
   |  //mode Multi Stream (MS)
13 |  z ← Wait_last_chan() ;       //attente de paquet à traiter en parallèle
14 |  Clear_instructions(z);
15 |  Edit_opcode(z,op_id,0) ;        //Edition instruction 0 du paquet voie z
16 |  NBOUT [z,0] ← 0;
17 sinon
18 |  NBOUT [x,j ] ← NBIN [x,j ] ;                       //mode Forward (FWD)
19 fin
```

Dans le cas du mode *Single Stream* (SSP), les données sont traitées directement par le PE. La modification consiste alors à modifier uniquement le champ `NB PASSES` en le décrémentant d'une itération (ligne 2). Si cette valeur devient nulle alors cela indiquera aux autres routeurs que la donnée a été complètement traitée pour cette opération.

Le mode *Single Stream & Forward* (SSF) est plus complexe car il est nécessaire de générer deux en-têtes. Une en-tête est transmise au routeur voisin avec les données entrantes non traitées et une autre en-tête est générée pour les données traitées par le PE. Pour le premier en-tête, le champ `NB PASSES` de l'instruction doit être réinitialisé à zéro afin que le paquet de donnée transmis puisse effectuer l'instruction suivant en parallèle (ligne 4). Pour le second en-tête, le champ `NB PASSES` est décrémenté d'une itération pour l'instruction concernée (ligne 6) et toutes les instructions en parallèle doivent être réinitialisées à zéro (boucle `tant que` en ligne 8). Cette réinitialisation assure la cohérence du flux de donnée sur la section traitée en parallèle. Il est à noter que la génération du second en-tête nécessite préalablement de récupérer la voie de communication choisie par l'arbitrage (fonction `Select_free_chan()` en ligne 5). C'est par cette voie que les données traitées par le PE vont transiter avec le second en-tête généré.

Pour le mode *Multi-Stream* (MS), un nouvel en-tête est généré pour être envoyé dans la voie de communication choisie (`z` ← `Wait_last_chan()` en ligne 13). Cet en-tête contient uniquement dans la première instruction l'identificateur de la dernière opération `op_id` à multiple entrées avec son nombre d'itération à zéro (`NBOUT[z,0]=0` en ligne 16). Toutes les autres instructions dans l'en-tête du paquet de données sortant sont nulles (fonction `Clear_instructions(z)` en ligne 14). Ainsi, ce mode doit clôturer le traitement de tout paquet de données. Il impose alors le positionnement de l'instruction dans l'en-tête dans ce type de mode.

La modification de l'en-tête n'est, de manière évidente, pas nécessaire dans le cas du mode *Forward* (FWD) qui correspond à `fmode=01`. Il s'agit simplement de transférer le paquet de données non-traité vers le routeur voisin.

5.4.3 Evaluation théorique des performances en temps

La figure 5.4 montre que les étapes de traitement d'un paquet peuvent se chevaucher temporellement. Ce chevauchement dépend de l'architecture et du choix d'implémentation du routeur. Ce choix doit avoir pour objectif de réduire au minimum la latence de traitement L_R d'un paquet de données dans le routeur esclave. Cette latence L_R correspond au nombre de cycle d'horloge entre l'entrée et la sortie du premier flit d'un paquet de donnée dans le routeur.

Le temps de traitement total d'un paquet de données T_t correspond au nombre total de cycles entre l'entrée du premier flit d'un paquet et la sortie du dernier flit de ce paquet. Ce temps correspond ainsi à celui de la traversée complète d'un paquet dans le routeur.

En considérant notre proposition d'architecture en figure 5.3, le temps de traitement total d'un paquet de données T_t est divisé en trois temps : La latence des interfaces L_i, la latence d'adaptation du routeur L_a et le temps T_{PE} de traitement des données du paquet par le PE tel que :

$$T_t = L_i + L_a + T_{PE} \tag{5.1}$$

La latence L_i correspond au nombre de cycles d'horloge total pour qu'un flit puisse traverser tous les ports nécessaires, dans un mode de fonctionnement défini dans un routeur esclave. Par exemple, pour le mode SSP, le flit doit traverser 4 ports : les ports d'entrée et de sortie du routeur dans le réseau et ceux permettant de communiquer avec l'unité de calcul. Ainsi, la latence L_i correspond à la somme totale des latences impactées par les interfaces du routeur avec le réseau (latence L_{br}) et les interfaces avec l'unité de calcul connectée au routeur (latence L_{bp}). La latence L_i est dépendante des modes du routeur au niveau fonctionnel, et de la taille des mémoires à chaque port d'interface au niveau architectural.

Par rapport aux étapes de traitement d'un paquet présentés en section 5.4.1, la latence d'adaptation L_a correspond au temps total, en nombre de cycles d'horloge, pour analyser l'en-tête, adapter le chemin de données interne avec si besoin un arbitrage, et mettre à jour cet en-tête. Sa valeur respecte alors l'équation 5.2.

$$L_a = T_a + (T_d - \delta_d) + (T_m - \delta_m) \tag{5.2}$$

La valeur de cette latence est variable et le chevauchement temporel entre les étapes est pris en compte par les valeurs δ_d et δ_m. Ces valeurs dépendent du mode de fonctionnement et de l'architecture du routeur esclave.

Par ailleurs, nous avons précédemment constaté au chapitre 2 que la majorité des opérateurs de traitement d'image sont orientés flux et peuvent être implémentés sous forme d'une structure pipelinée d'unités de calcul. En supposant que les unités de calcul rattachées au routeur *esclaves* ont une structure interne pipelinée de latence L_{PE}, la valeur du temps de traitement T_{PE} d'un paquet constitué de n flits, respecte alors l'équation 5.3 :

$$T_{PE} = n + L_{PE} \tag{5.3}$$

Dans ces conditions, la latence totale du routeur L_R équivaut ainsi à :

$$L_R = L_i + L_{PE} + L_a \tag{5.4}$$

Le choix d'implémentation du routeur doit ainsi optimiser la valeur L_R afin que la latence totale entre deux routeurs maîtres soit la plus réduite possible, selon le nombre de routeur esclaves les séparant.

5.5 Implantation matérielle : *Data Flow Router*

Nous proposons à présent un choix d'implémentation sur puce du routeur esclave que nous appelons *Data Flow Router* (DFR). Dans cette section, nous présentons dans un premier temps, une vue globale de l'architecture avec une description de ses interfaces et des méthodes de communication. Nous détaillons par la suite les sous-modules qui composent ce routeur : les unités d'aiguillage de paquets de données et le contrôleur global interne du routeur. Pour finir, nous terminons par une discussion de cette proposition en terme d'adaptabilité pré-synthèse et post-synthèse.

5.5.1 Architecture globale du *Data Flow Router*

Une vue globale de l'implémentation du DFR est présentée en figure 5.5.

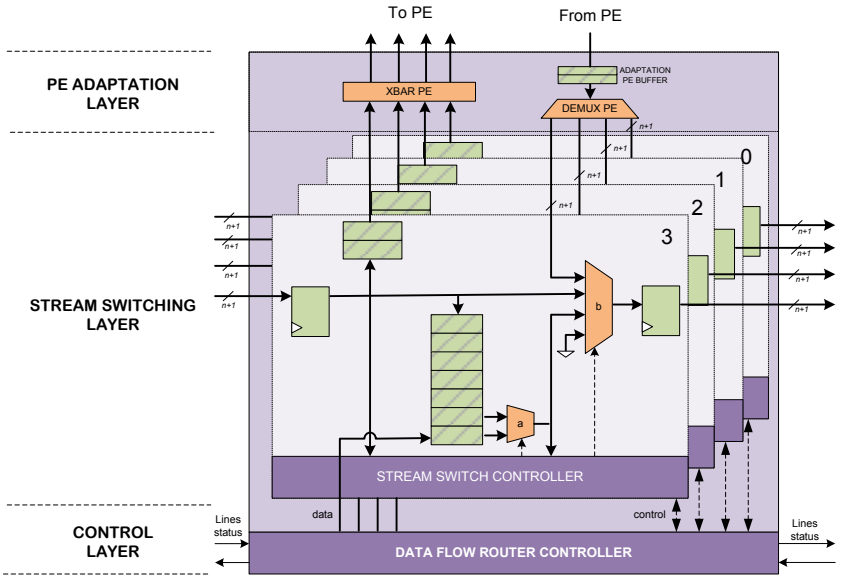


FIGURE 5.5: Architecture globale du DFR à quatre voies (k=4)

Cette proposition d'implémentation est organisée en trois couches : une couche centrale dédiée à l'aiguillage des paquets de données (*stream switching layer*), une couche d'adaptation vers le PE relié au routeur (*PE adaptation layer*) et une couche de contrôle global interne (*control layer*).

La couche centrale d'adaptation est constituée de plusieurs unités de commutation de flux qui sont implémentés par des modules appelées *Stream Switch*. L'implémentation respecte l'indépendance de ces unités entre chaque voie de communication comme spécifié dans la proposition présentée en figure 5.3. Il existe alors autant d'unités *Stream Switch* que de voies de communication.

Cette couche est en intéraction avec la couche de contrôle contenant le contrôleur global interne du routeur (*DFR Controller*). Le rôle principal de cette unité est de contrôler les adaptations des chemins de données et d'arbitrer les différentes requêtes des différents *Stream Switch*.

En fonction des décisions d'arbitrage et des modes de fonctionnement utilisés, la couche de contrôle va commander la couche d'adaptation vers le PE afin de rediriger correctement les paquets de données. Cette couche d'adaptation au PE contient les unités de

sélection de flux A et B de l'architecture qui sont implémentées respectivement par un crossbar et un démultiplexeur.

5.5.2 Interfaces de communication du routeur

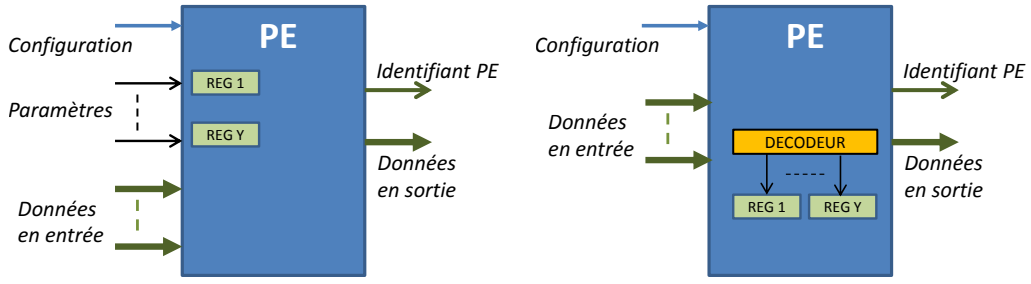
5.5.2.1 Interfaces de communication routeur-PE

Le routeur dispose de plusieurs ports permettant de communiquer avec les routeurs esclaves voisins et avec le PE connecté. Nous rappelons que tout PE connecté au réseau possède un ou plusieurs ports d'entrées de données selon le type d'opération et un port unique en sortie pour fournir le résultat de l'opération.

Les PEs peuvent nécessiter d'être configurés par des paramètres spécifiques pour effectuer une opération. Ces paramètres peuvent être des coefficients de calcul ou des modes d'utilisation particuliers.

Selon le choix d'implémentation du PE, les méthodes de chargement des paramètres de configuration peuvent être de deux types : en *série* ou en *parallèle*.

La figure 5.6 illustre les deux types de PE que nous pouvons connecter au réseau.



(a) PE en configuration parallèle (b) PE en configuration série

FIGURE 5.6: Configuration des paramètres d'un PE

Le premier type de configuration, illustré en figure 5.6(a), permet de charger en parallèle des paramètres du PE qui sont stockés dans des registres dès que le signal de configuration est actif. Le PE dispose ainsi d'un port de communication dédié pour chaque paramètre à configurer.

Le deuxième type de configuration, illustré en figure 5.6(b), utilise directement les canaux de communication des données pour transmettre les paramètres de configuration en série, par le biais d'un paquet de configuration spécifique.

La figure 5.7 illustre un exemple de paquet permettant de configurer un PE dans le réseau. La distinction entre un paquet de données pixel et un paquet de données de configuration des registres, est faite au niveau d'un bit de configuration dans le champ de l'en-tête dédié aux attributs. Dès qu'un paquet de configuration de l'unité de calcul est détecté, un signal spécifique de configuration indique au PE une demande de chargement de nouveaux paramètres.

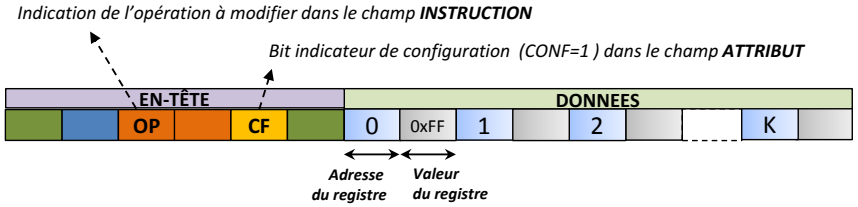


FIGURE 5.7: Paquet de configuration des unités de calcul (PEs)

Ce type de paquet permet de configurer un ensemble de registres stockant les paramètres de configuration du PE. Le contenu de ce paquet contient de façon alternée la position du paramètre à configurer et la valeur à attribuer. Le paquet de configuration nécessite ainsi d'être décodé à l'intérieur du PE.

Les figures 5.8 et 5.9 décrivent respectivement les interfaces du routeur avec un PE pour ces deux types de chargement.

Dans le premier cas illustrant une configuration série dans la figure 5.8, les informations de configuration sont directement envoyées sur un des canaux de communication de taille $n+1$ vers le PE. En effet, à chaque port d'entrée ou de sortie de données de taille n, est associé un signal de validité qui accompagne la donnée transmise afin d'être prise en compte par le PE.

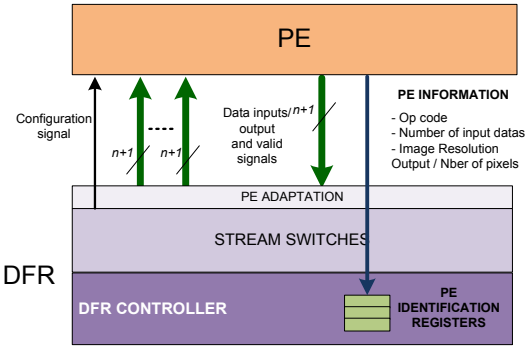


FIGURE 5.8: Interfaces du DFR avec le PE - configuration série

Notons que le temps de décodage peut prendre plusieurs dizaines de cycles d'horloge selon le nombre et la taille des paramètres.

Le second cas est illustré par la figure 5.9. Nous constatons que cette solution nécessite, dans notre routeur esclave, une spécialisation de la couche d'adaptation, afin de désérialiser temporellement les paramètres. Cette couche doit ainsi être complétée par des registres, appelés `CONFIG REGISTERS` sur la figure, permettant ce type de chargement.

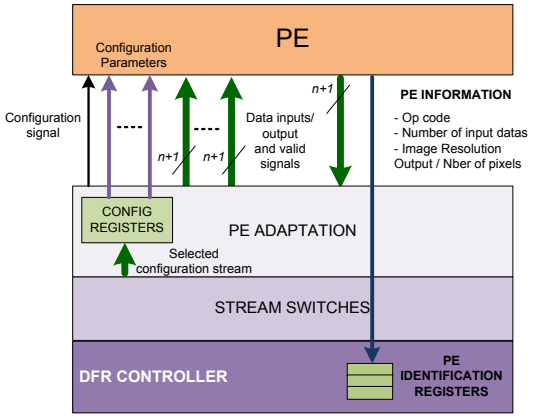


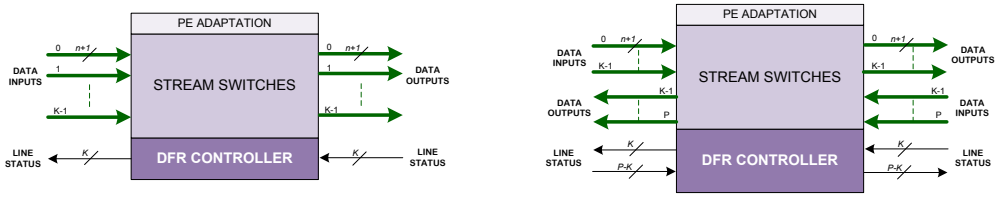
FIGURE 5.9: Interfaces du DFR avec le PE - configuration parallèle

En résumé, les deux méthodes de chargement présentent des avantages et des inconvénients. Un chargement des paramètres en série permet d'économiser la surface du routeur mais présente l'inconvénient de devoir spécialiser le PE avec un décodeur interne pour extraire les paramètres. Une partie du PE devient ainsi spécifique au réseau. A l'inverse, le chargement des paramètres de configuration en parallèle permet au PE de rester indépendant du réseau et d'être réutilisable pour d'autres types d'implantation dédiées dans d'autres architecture par exemple.

5.5.2.2 Interfaces de communication inter-routeur

Les routeurs communiquent les paquets avec un signal de validité de la donnée. Dans le cas du routeur esclave, une ou plusieurs voies de communication peuvent être occupées selon les modes de fonctionnement. L'état d'utilisation d'une voie de communication est transmis de manière *asynchrone* entre les routeurs esclaves qui séparent deux noeuds maîtres. La figure 5.10 illustre le cas mono-directionnel avec un seul signal de synchronisation et le cas bi-directionnel avec deux signaux de synchronisation dans les deux sens de cirulation.

Pour ces deux cas, un signal appelé *line status*, permet d'informer au routeur esclave voisin de l'occupation d'un canal de communication, afin de réaliser une méthode de



(a) mono-directionnelle (b) bi-directionnelle

FIGURE 5.10: Interfaces inter-routeur

contrôle de flot de données de type *on/off*. Ce signal dit de type *backpressure*, remonté au routeur *maître* émetteur, permet de bloquer l'émission des flits de données afin d'éviter une collision des données.

5.5.3 Architecture du *Stream Switch*

L'unité, que nous appelons *Stream Switch*, représente l'unité d'adaptation locale du chemin de données dans un routeur DFR. La conception du *Stream Switch* est importante. Elle contient le mécanisme d'analyse de paquet et sa structure est dupliquée proportionnellement en fonction du nombre de voies de communication définies pour un routeur. Ce nombre de voies impacte directement la surface d'implémentation requise ainsi que la complexité du contrôleur global, étant donné que le contrôle est établi de manière hiérarchique.

5.5.3.1 Description de l'architecture

L'architecture détaillée du *Stream Switchs* est présentée en figure 5.11.

Cette architecture contient un contrôleur local, appelé *Stream Switch Controller*, des unités de mémorisation, implémentées par des registres, et des unités de modification de chemin de données, implémentées par les multiplexeurs *a* et *b*. Ces multiplexeurs sont gérés par le *Stream Switch Controller*.

Nous distinguons quatre catégories de signaux : les données (avec leur signal de validité), les signaux de commande, les signaux de protocole de communication (acquittement) et des signaux d'information (indicateur d'état, données décodées, etc.).

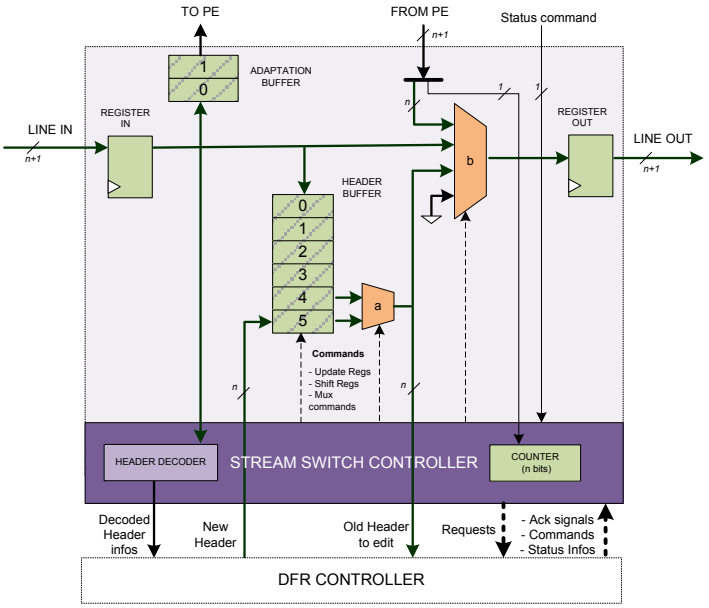


FIGURE 5.11: Architecture du Stream Switch

Nous notons que la capacité de mémorisation requise est directement liée à la taille en flits fixée pour l'en-tête des paquets de données. En effet, l'unité de mémorisation principale, représentée par les 6 registres sur le schéma 5.11, est celle stockant les données de l'en-tête. Ce stockage est nécessaire pour l'analyse et l'édition des champs de l'en-tête en parallèle. Ces données sont ensuite relues séquentiellement afin d'être transmise dans le réseau. Les autres registres sont utilisés pour tamponner les données aux interfaces de chaque port d'entrée et de sortie du routeur.

Dans l'architecture proposée pour le *Stream Switch*, l'en-tête est la partie qui génère une latence importante par son décodage et son analyse. Afin d'accélérer le décodage et le calcul du chemin de donnée, une solution proposée par Lee [145] dans un réseau sur puce, consiste à définir deux horloges (une pour la donnée et une pour l'en-tête) afin d'accélérer le transfert des données utiles qui ne nécessitent pas d'analyse. En effet, le décodage de l'en-tête, défini dans le routeur, impacte la fréquence maximale de fonctionnement du routeur. En introduisant, deux chemins différents, on serait capable de maximiser la fréquence de transfert des paquets. Cependant, dans notre proposition de travailler avec des structures de paquet de données de taille importante, la latence d'analyse de l'en-tête devient très inférieure au temps de transfert des données à traiter.

5.5.3.2 Contrôle local : *Stream Switch Controller*

La figure 5.12 détaille le contenu du *Stream Switch Controller*.

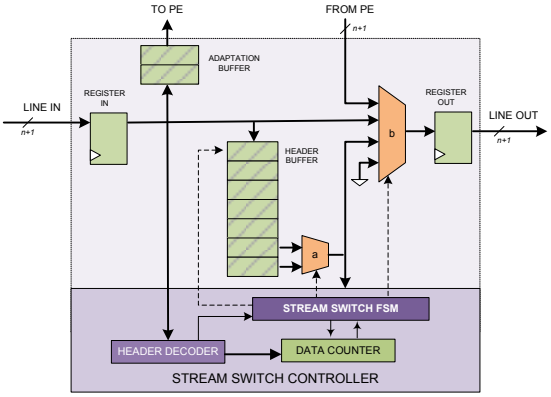


FIGURE 5.12: Structure interne du *Stream Switch Controller*

Celui-ci est implémenté à l'aide d'une machine à états, appelée *Stream Switch FSM*, couplée avec un compteur de données interne, appelé *Data Counter* et une unité de décodage de l'en-tête, appelée *Header Decoder* sur la figure 5.12. Ce dernier permet de décoder à la volée les en-têtes de paquets entrants afin de précharger différents registres comme celui du compteur interne. Ce compteur permet de décompter le nombre de flits contenus dans un paquet de donnée. Il est ainsi utilisé pour déterminer la fin de transmission d'un paquet de données complet. Nous pouvons remarquer que ce compteur n'est utilisé que dans deux cas de modes de fonctionnement qui ne mobiblisent qu'une seule voie, à savoir le mode *Forward* (FWD) et le mode *Single Stream* (SSP). Pour les autres modes de fonctionnement qui utilisent plusieurs voies d'un routeur, un second compteur implémenté dans le contrôleur global, appelé *DFR controller*, est utilisé pour décompter les flits des paquets provenant de l'unité de calcul.

La figure 5.13 illustre la machine d'état simplifiée, implémentée dans le *Stream Switch Controller*.

La première étape, dans l'état `ANALYSIS`, correspond à l'analyse de l'en-tête d'un paquet de données. Cet état est lié au phases de décodage des instructions, de chargement des registres et de la prise de décision du mode de fonctionnement selon les informations dans l'en-tête.

Selon la capacité du PE à réaliser l'opération souhaitée sur la donnée du paquet entrant, le contrôleur se retrouve soit dans le mode de fonctionnement *Forward* consistant à

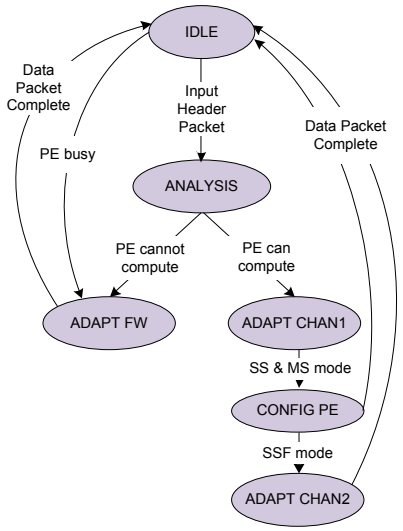


FIGURE 5.13: Machine à états du *Stream Switch Controller*

transférer le paquet sans aucune adaptation, dans l'état `ADAPT FW`, ou soit un autre mode qui adapte le chemin de données interne afin d'envoyer les données du paquet vers le PE, dans l'état `ADAPT CHAN1`. Une autre étape supplémentaire est nécessaire, dans l'état `ADAPT CHAN2`, pour des adaptations multi-voies.

Nous notons que le mode *Forward* peut être déclenché de manière directe, dans l'état `ADAPT FW`, si le PE est déjà occupé, grâce à un signal *PE busy* provenant de celui-ci. Au niveau de l'architecture interne, les paquets de données entrants sont alors directement envoyés vers la sortie, comme illustré par la figure 5.14. Le chemin de donnée selectionné, sur lequel transite le flux de données, est tracé par une surimpression rouge sur ce schéma.

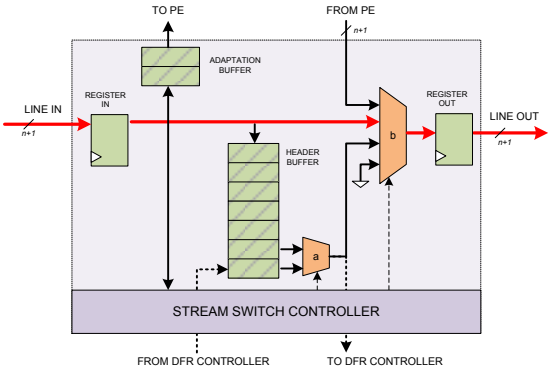


FIGURE 5.14: Mode d'exécution *Forward* automatique

La figure 5.15 illustre le chemin de donnée utilisé dans le cas de l'analyse de l'en-tête. Nous voyons sur cette figure que l'en-tête est mémorisé dans des registres dédiés pour décoder les instructionss.

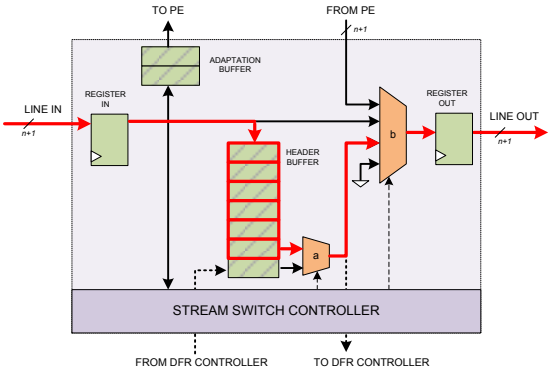
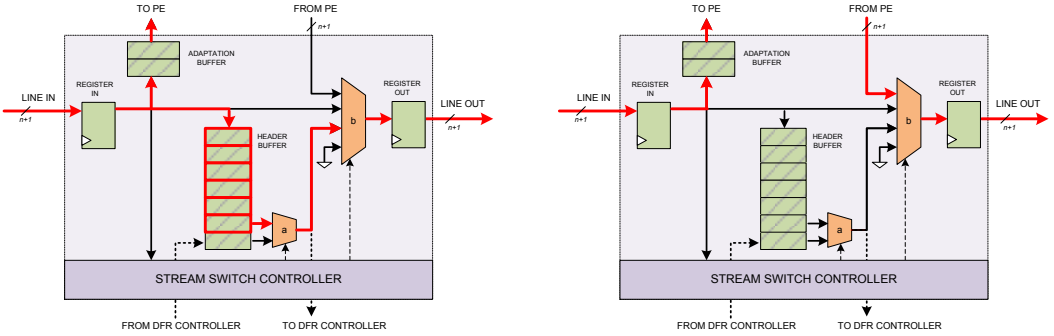


FIGURE 5.15: Mode d'exécution *Forward* avec analyse de l'en-tête

Après analyse de cet en-tête, si aucune opération demandée ne peut être exécutée par le PE, alors le paquet de données complet est transmis au routeur voisin suivant en mode *Forward* (FWD).

Au contraire, si le PE est capable de réaliser l'opération demandé et que ce dernier est libre (signal *PE Busy* à 0), alors le routeur peut activer le mode SSP, SSF ou MS. Le cheminement du paquet est alors le stockage de l'en-tête puis l'envoi des données au PE, à l'état `ADAPT CHAN1`. L'en-tête mémorisé est édité au cours du traitement des données par le PE, dans l'état `CONFIG PE` puis il est envoyé dès la première donnée disponible en sortie du PE. La figure 5.16(a) illustre cette première étape de transmission de l'en-tête.



(a) SS : Transfert de l'en-tête

(b) SS : Transfert de la sortie du PE

FIGURE 5.16: Mode d'exécution *Single Stream*

Dans le cas du mode *Single Stream* (SSP), les flits de données en sortie du PE suivent directement l'en-tête, comme illustré par la figure 5.16(b). Le mode *Multi-Stream* (MS) est similaire sauf que les données en sortie du PE ne sont disponibles que lorsque toutes les données d'entrée en parallèle du PE sont arrivées.

Le mode *Single Stream & Forward* (SSF) est plus complexe car il nécessite une seconde voie disponible pour transmettre les données en sortie du PE, dans l'état `ADAPT CHAN2`. Comme remarqué précédemment, un nouvel en-tête doit être généré par le contrôleur global, *DFR controller*, pour les nouvelles données issues du PE. La figure 5.17 illustre ce passage de l'en-tête qui est édité dans le contrôleur global avant d'être transmis dans la voie de communication sélectionnée.

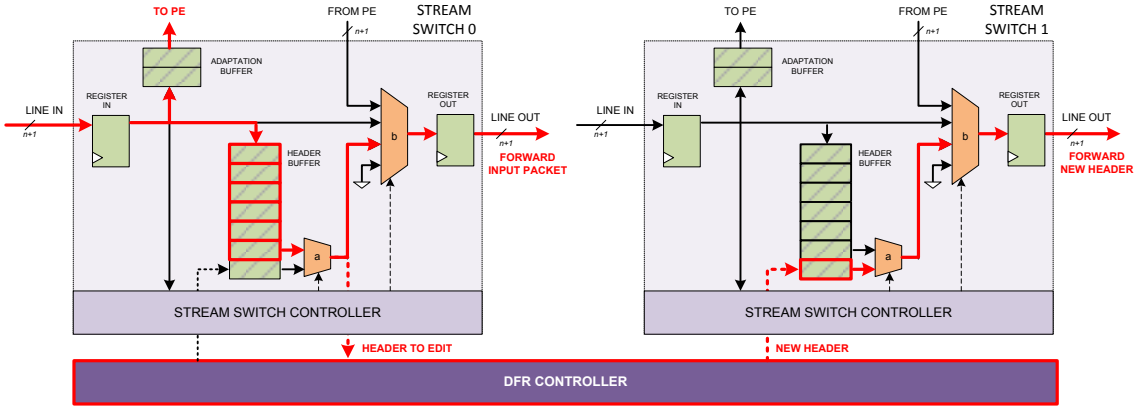


FIGURE 5.17: Mode d'exécution *SSF* : Transfert de l'en-tête

Par la suite, la figure 5.18 illustre la transmission de deux flux de sortie en parallèle : le flux de donnée entrant non traité et le flux de données traité par le PE. Dans cet exemple, le *Stream Switch* 1 selectionné, récupère le nouvel en-tête du contrôleur global et le transmet suivi des données traitées en sortie du PE.

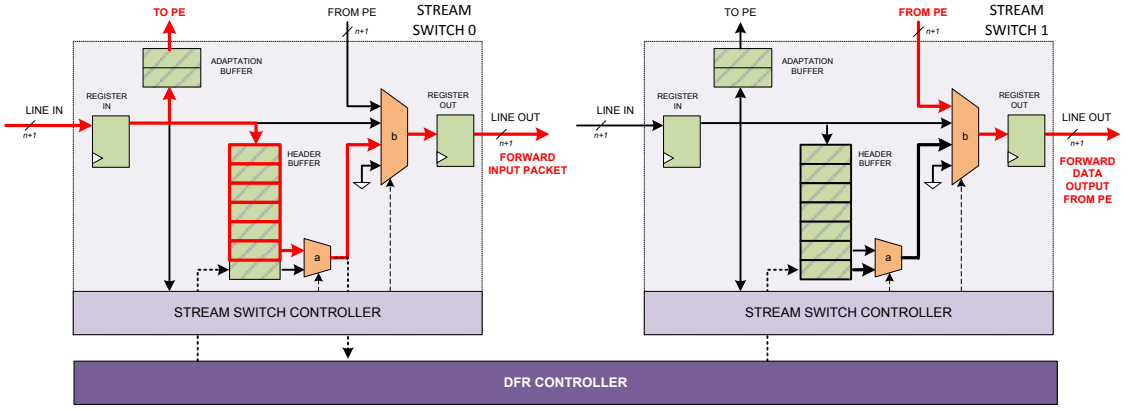


FIGURE 5.18: Mode d'exécution *SSF* : Transfert des données du PE

5.5.4 Description du *DFR Controller*

Le *DFR Controller* reçoit en permanence des requêtes d'accès au PE avec un mode de fonctionnement imposé de manière implicite par les instructions. Nous rappelons que le

rôle principal de cette unité est de réaliser l'arbitrage entre ces différentes requêtes et de commander la couche d'adaptation d'accès au PE. Nous avons également noté que selon les modes de fonctionenment activés, cette unité est capable d'éditer un nouvel en-tête, afin d'assurer une cohérence temporelle dans l'exécution des opérations sur la donnée.

La figure 5.19 détaille l'architecture de ce contrôleur. Cette architecture contient une machine à états, appelée `DFR CTRL FSM`, qui pilote des unités de décodage et d'encodage d'en-tête, appelées `Header Decoder` et `Header Encoder` ainsi qu'un compteur de pixels, appelé `Data Counter`. De manière identique au *Stream Switch Controller*, cette unité est utilisé pour décompter les données en sortie du PE.

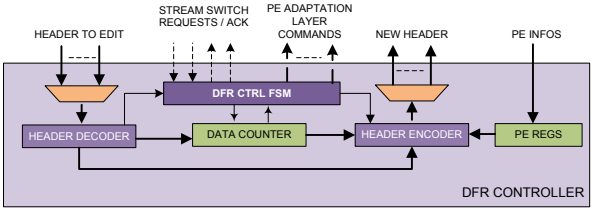


FIGURE 5.19: Architecture du *DFR Controller*

La machine à états, présentée en figure 5.20, résume le fonctionnement de la machine à état. Elle reprend de manière détaillée l'algorithme 3 proposé pour l'évolution des états du contrôleur auquel nous ajoutons les états `SET COUNTERS` et `EDIT HEADER` pour charger les registres de comptage et éditer l'en-tête.

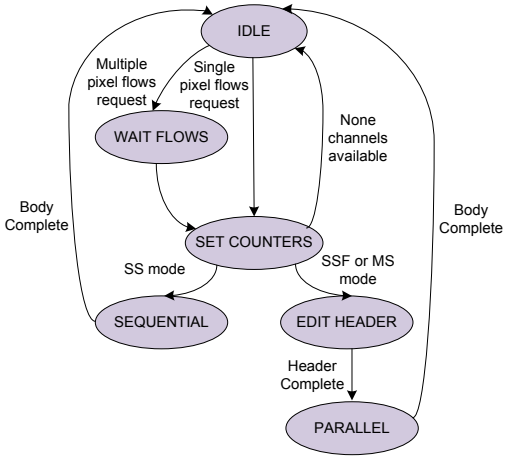


FIGURE 5.20: Machine à états du *DFR Controller*

Pour éviter tout risque de collision ou perte de paquets de données, une connaissance permanente de l'occupation des voies de communication est impérative entre les routeurs esclaves voisins. Cette information d'occupation est transmise sur tous les contrôleurs globaux des routeurs esclaves entre deux routeurs maîtres, par des signaux asynchrones,

appelés *lines status*, pour chaque voie de communication. Ces signaux sont alors transmis simplement dans un routeur esclave en utilisant une porte `OU` logique entre les signaux d'occupation *lines status* des routeurs voisins et ceux des *Stream Switch* connectés au contrôleur global.

Le *DFR Controller* intègre la méthode d'arbitrage, décrite dans la section précédente, qui respecte une priorité à indice selon un numérotage pré-défini des voies. La machine à états, en figure 5.20, ne présente que le traitement des requêtes si le PE est disponible. Dans le cas où le PE est déjà occupé, toutes les autres requêtes sont rejetées jusqu'à ce que l'intégralité du paquet de données, envoyée au PE, a fini d'être traitée. Dans cette situation précise, les *Stream Switchs* sont alors automatiquement adaptés en mode de fonctionnement *Forward* (FWD).

5.6 Synthèse et comparaison de la proposition

L'architecture du DFR diffère des architectures de routeur de NoC plus courant dans la littérature comme Hermes [114], par la structure des paquets utilisée, par la méthode d'adaptation du chemin de donnée et par les modes de fonctionnement inédits utilisant plusieurs voies de communication.

5.6.1 Structure des paquets

La majorité des routeurs de données sur puce sont définis pour traiter des paquets avec un faible nombre de flits et appliquer un routage déduit de l'adresse du noeud de destination [14]. Selon le type d'algorithme de routage utilisé, l'en-tête contient essentiellement des informations sur le type de données et sur les adresses, contrairement à notre proposition qui rajoute un champ dédié aux instructions.

Le tableau 5.2 compare les paquets à transmettre dans le cas d'utilisation d'un routeur DFR par rapport à une solution de routeur (R) plus fréquente dans la littérature [14].

Pour des applications orientées flux de données et en particulier dans le domaine de la vision, les messages qui circulent dans le réseau sont principalement des trames. Les opérations requises s'appliquent de façon redondantes sur un groupe de pixels de la taille d'une ligne ou d'une image complète. Ainsi, le volume de données à transmettre entre

	structure paquet		contenu de l'en-tête			taille totale données (bits)
	en-tête	donnée	adresse	donnée	commande	
R	court	court/moyen	oui	oui/non	non	$S_H * \lceil M/d \rceil + M$
DFR	moyen	long	oui	oui	oui	$S_H + M$

TABLE 5.2: Comparaison DFR avec un routeur NoC plus fréquent (R) sur les paquets de données à traiter

le noeud émetteur et le noeud destinataire est dépendant de la taille de l'en-tête et du fractionnement du message trame en plusieurs paquets.

Soient M la taille en flits d'un message représentant une image complète à transmettre, S_H la taille définie pour l'en-tête de paquet de données dans le réseau et d la taille fixe en flits définie pour la donnée utile transportable dans un paquet.

Pour une taille de paquet fixe, le volume de données V à transmettre sera $V = S_H * \lceil M/d \rceil + M$. Dans le cas où la taille du paquet de données est égale à une trame complète alors le volume $V = S_H + M$. Le surcoût est alors dépendant de la taille de l'en-tête.

5.6.2 Méthode et latence d'adaptation du chemin de données interne

La majorité des routeurs sur puce traitent de manière séquentielle les différents paquets de données entrants suivant un arbitrage défini selon la topologie et le type de données. Cette méthode implique d'une part une latence de traitement des paquets qui est proportionnelle au nombre de paquets simultanés à gérer, avec une nécessité de bufferiser les paquets en attente. La bufferisation nécessaire augmente en fonction du nombre de paquets souhaitant accéder à un même port de sortie et elle peut être compensée par la mise en place d'un algorithme de routage permettant de rediriger des paquets vers d'autres chemins.

En considérant la structure de paquet de données choisie, il est important de minimiser les besoins en stockage pour chaque paquet de données entrant. Nous faisons ainsi le choix dans notre architecture de réduire le degré de liberté pour chaque paquet entrant de manière à être capable de maintenir des voies de communication indépendantes et unidirectionnelle, afin de minimiser le stockage et réduire les latences de traitement. Des paquets peuvent ainsi être traité en parallèle sans attente supplémentaire car chaque voie est capable de s'adapter localement. Le DFR est néanmoins également capable de s'adapter globalement sur plusieurs voies dans le cas de mode fonctionnement avancé.

Le tableau 5.3 compare les méthodes d'adaptations entre un routeur de conception plus fréquent et le DFR proposé.

	Adaptation			
	niveau	traitement des paquets	latence	communication contrôle
R	global	séquentiel	moyenne-élevée	paquet séparé des données
DFR	local/global	parallèle	faible-moyenne	intégré aux données

TABLE 5.3: Comparaison DFR avec un routeur NoC plus fréquent(R) sur l'adaptation du chemin de données interne

Dans notre contexte, la solution de routeur conventionnel implique d'une part la mise en oeuvre d'un ou plusieurs modules de contrôle dont le rôle est de séquencer les paquets de données pour réaliser l'application. Cette solution engendre ainsi une latence supplémentaire pour la communication des commandes et des instructions avant d'envoyer les données à traiter. Notre proposition de combiner données et instructions nous permet alors de d'éviter cette latence et d'assurer une cohérence des opérations sans impliquer des méthodes de contrôle complexes ou des surcouches logiques de réordonnancement de paquets, en particulier dans les interfaces réseaux entre le PE et le routeur.

5.6.3 Micro-architecture interne et interfaces PE

Comme discuté précédemment, la micro-architecture du DFR est dépendante du nombre de voies de communication à gérer en parallèle. Ce nombre impacte la consommation en surface, du fait de sa capacité à gérer localement l'adaptation de chemin de données et de son contrôle hiérarchique. Plus fréquemment, dans un routeur, le contrôle global est au contraire géré plus simplement de manière unique par une unité centralisée.

Le tableau 5.4 résume les différences entre un routeur plus fréquent (R) et le DFR par rapport à l'architecture interne et l'interfaçage de l'unité de calcul (PE).

	contrôle	NI	modes	surface	PE(SC)	PE(ME)
R	unique centralisé	séparée	FWD, SS	faible-moyenne	non	non
DFR	hiérarchique	intégrée	FWD, SS, SSF, MS	moyenne-elevée	oui	oui

TABLE 5.4: Comparaison sur l'architecture interne et l'interfaçage du PE

L'architecture du DFR favorise les solutions d'architecture pipelinées avec des PEs orientés flux de données. En particulier, il autorise l'interfaçage de PE correspondant à des IPs dédiés ne contenant aucun contrôle interne (`PE(SC)` dans le tableau). Il permet, en

outre, d'interfacer à moindre effort un PE conçu pour des chaînes de traitement dédiées. Le DFR autorise également l'utilisation de PE à multiple entrées (`PE(ME)` dans le tableau) permettant ainsi de combiner deux paquets de données représentant deux flux de données différents grâce au mode de fonctionnement *MS*.

L'interface réseau `NI` est également simplifiée dans le DFR et intégrée directement dans les *Stream Switches* qui se chargent de décoder et d'éditer les en-têtes de paquets.

En ce qui concerne les modes de fonctionnement, un routeur se charge, en général, principalement soit de rediriger vers un port de sortie à la manière du mode *Forward* ou soit de traiter le paquet par le PE à la manière du mode *Single Stream* comme dans les routeurs proposés dans [130, 142]. Le DFR autorise, par son architecture, les modes *Single Stream & Forward* et *Multi-Stream* permettant de réaliser de manière efficace un assemblage de PEs de type pipeline-parallèle et d'utiliser des PEs traitant des paquets en parallèle.

En résumé, l'architecture du DFR proposée est certe relativement plus coûteuse en surface par rapport à des solutions plus classiques, mais est particulièrement efficace dans le traitement en parallèle de plusieurs paquets de données. Ce coût est compensé par sa capacité à adapter son chemin de données dynamiquement afin de traiter des paquets efficacement par la combinaison donnée-instruction dans le même paquet.

5.7 Prototypage et Evaluation

Afin de valider le fonctionnement et évaluer les performances en temps et surface, la micro-architecture proposée pour le DFR a été implémentée dans un FPGA (Altera Stratix III EP3SL150). Dans ce but, nous utilisons le flot de développement standard avec une description RTL en VHDL avec une approche modulaire décrivant tous les composants. L'architecture est décrite de façon à être paramétrable avant synthèse (adaptabilité pré-synthèse) en terme de nombre de ports et taille de bus.

Pour cette implémentation, nous avons fixé le nombre de voies à quatre (k=4) avec une taille de bus de 32 bits (n=32). Cela équivaut donc à quatre ports d'entrée et quatre ports de sortie auxquels s'ajoutent les ports d'entrée et sortie du PE. Ce paramétrage permet de réaliser entre deux noeuds *maîtres*, deux voies de communication dans deux sens de

circulation. Cette configuration permet ainsi aux deux noeuds maîtres d'être émetteurs et de définir des instructions permettant l'adaptation du chemin pour les deux sens du flux de données, particulièrement pour les modes SSF et MS. Une taille de bus de 32 bits est suffisante pour pouvoir véhiculer différents types de granularité de pixel, allant d'une image monochrome 8 bits vers une image couleur 24 bits par exemple.

5.7.1 Implémentation de l'en-tête

Dans le modèle de réseau, nous pouvons définir que la taille du phit est égale à celle du flit, donc la taille de bus choisie fixe également la taille d'un flit de données. La structure d'en-tête proposée et illustrée en figure 4.15, implique une taille minimum équivalente à cinq flits de données.

Le tableau 5.5 présente une proposition des tailles de chaque champ dans l'en-tête : les champs de début et fin d'en-tête de taille S_t, le champ du nombre de pixel de taille S_p, le champ des instructions de taille S_i et le champ des attributs de taille S_a. Nous proposons un en-tête de taille 192 bits répartie en 6 flits de données. Nous fixons également quatre instructions de taille 16 bits associées à chaque paquet de données.

	S_t	S_p	S_i	S_a	S_H (total)
Taille (flits)	1	1	2	1	6
Taille (bits)	32	32	64	32	192

TABLE 5.5: Partionnement des champs de l'en-tête sur 6 flits

Le détail de ce partitionnement est résumé dans le tableau 5.6. Les flits considérés comme des balises de début et de fin de l'en-tête sont détectés avec la valeur 0xFFFFFFFF. Le champ attributs contient à la fois les informations d'adressage (source, destination) et les informations caractérisant la donnée (type de capteur, temps index). Il contient un bit permettant de détecter le type de données transporté dans le paquet permettant de distinguer les paquets de données pixel avec d'autres types de paquet.

Une proposition de découpage d'une instruction en quatre champs (numéro de la ligne d'instruction, identification de l'opérateur, nombre d'itération, traitement de l'opération en parallèle) sur 16 bits est présentée dans le tableau 5.7.

Pour des instructions de taille 16 bits, ce découpage permet de traiter au maximum 16 lignes d'instruction soit 64 opérations distinctes en considérant des en-têtes contenant 4

N° Flits (32 bits)	Contenu (taille en bits)
0	Header Start (0xFFFFFFFF) [31..0]
1	Nombre de pixels [31..0]
2	Instruction 0 [31..16], Instruction 1 [15..0]
3	Instruction 2 [31..16], Instruction 3 [15..0]
4	Bit config [12], ID [11..8], Timestamp [7..4], Noeud Maître Source [3..2], Noeud Maître Destination [1..0]
5	Header End (0xFFFFFFFF) [31..0]

TABLE 5.6: Détail du contenu de l'en-tête (flits de 32 bits)

Champ	Taille (bits)	Position	Information
INST NUMBER	4	[15..12]	Numéro de la ligne instruction
PE OPCODE	6	[11..6]	Type de PE requis
NB PASS	4	[5..2]	Nombre de passe requis
TAG	2	[1..0]	Indication de traitement en parallèle

TABLE 5.7: Structure d'une instruction sur 16 bits

instructions. Chaque opération peut être réalisée avec un nombre maximum de 16 itérations consécutives. Nous rappelons que le champ *TAG* permet d'identifier la possibilité de traiter en parallèle ou en séquentiel les opérations contenus dans une ligne instruction.

5.7.2 Evaluation en surface du routeur

Le tableau 5.8 détaille la consommation en surface FPGA (Stratix III EP3SL150) du routeur DFR avec la structure de l'en-tête proposée de 6 flits. Le paramétrage de ce routeur est de 4 voies de communication pour une taille de bus de 32 bits.

	Logiques	Registres	Mem (bits)	(% FPGA)
Stream Switch	294	246	100	0.25
DFR Controller	226	130	0	0.2
Buffer sortie PE	21	10	224	0.01
DFR (total)	1154	1738	624	1.5

TABLE 5.8: Surface : Data Flow Router (EP3SL150)

Nous obtenons une consommation totale inférieure à 2 % du FPGA avec une fréquence maximum de 234 MHz. Cette surface est raisonnable dans cette cible afin de pouvoir implémenter des applications de vision utilisant une dizaine d'opérateurs à grain moyen.

5.7.3 Evaluation de la latence du routeur

Pour l'évaluation de la latence totale L_R du routeur implémenté, nous introduisons deux paramètres non déterministes : la latence δ_h en cycles d'horloge entre la donnée et l'en-tête, et la latence δ_T qui sépare deux paquets de données nécessitant d'être combinés dans un PE.

Nous mesurons donc la latence L_R selon les différents modes de fonctionnement du routeur avec le paramétrage présenté précédemment. La figure 5.21 présente un exemple dans le cas de deux voies de communication implémentée dans un routeur. La voie 0 introduit un paquet de donnée dont l'analyse déclenche une requête d'accès au PE connecté au routeur. Ce PE étant libre, le *Stream Switch* associé à la voie s'adapte alors en mode *Single Stream* avec une latence de traitement $L_R(SS)$. Au cours de ce traitement, un autre paquet intervenant dans la voie 1 est automatiquement transférée en sortie avec une latence $L_R(F)$ sans analyse car le PE n'a pas encore fini de traiter la paquet de la voie 0.

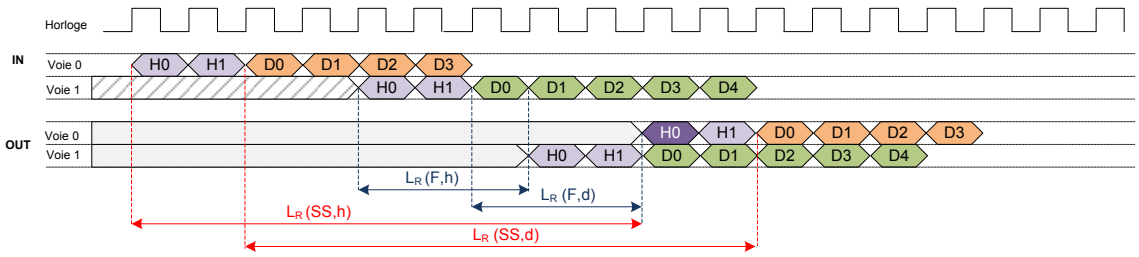


FIGURE 5.21: Chronogramme du mode *Forward* et *Single Stream*

Le tableau 5.9 présente les latences mesurées en fonction des différents modes de fonctionnement sans compter le temps d'arbitrage équivalent à 1 cycle supplémentaire.

Mode	L_a (cycles)	L_i (cycles)	L_R (cycles)
Forward (auto)	0	2	2
Forward (check)	5	2	7
Single Stream	6	6	$12+L_{PE}+\delta_h$
SSF (paquet 1)	5	2	7
SSF (paquet 2)	6	6	$12+L_{PE}+\delta_h$
MS	$6+\delta_T$	6	$12+L_{PE}+\delta_T$

TABLE 5.9: Latences d'adaptation et totale selon les modes de fonctionnement

Dans le premier cas, il s'agit du mode *Forward* activé automatiquement dès qu'un PE est déjà occupé. Ce mode correspond concrètement à un bypass dans le routeur et assure

une latence de traversée minimale qui est ici équivalente à deux cycles. Comme décrit précédemment, ce mode peut être aussi activé après analyse du paquet de données dans le cas où celui-ci ne peut pas être traité par le PE raccroché. Une latence d'analyse de 5 cycles s'ajoute alors aux 2 cycles correspondant aux interfaces.

Pour le mode *Single Stream*, la latence d'adaptation est de 6 cycles et contient la requête de modification du chemin de donnée de 1 cycle. Les latences de bufferisation sont plus importantes pour adapter le chemin de données. La latence totale tient alors compte de celle du PE et d'un temps supplémentaire (pouvant être nul) correspondant à l'écart en nombre de cycles entre l'arrivée des flits d'en-tête et de données.

Le mode *Single Stream & Forward* génère deux paquets de données : un transmis directement sans traitement et un autre traité. Le paquet non traité (paquet 1 dans le tableau) possède la même latence que dans le cas de figure du mode *Forward* avec analyse, à savoir 7 cycles et le second avec une latence équivalent au mode *Single Stream*. Nous rappelons que ce mode mobilise deux voies de communication. Il n'y a donc pas de nécessité d'arbitrer les flits de données en sortie d'une seule voie de communication ce qui assure une bande passante maximale pour le flux traité et non traité.

Le mode *Multi-Stream* possède une latence naturellement dépendante de l'écart en cycle δ_T entre l'arrivée des paquets de données nécessitant d'être combinés dans le PE.

Nous rappelons enfin que dans le cas d'application utilisant des opérateurs orientée flux de données, les latences de traversée de routeurs deviennent quasiment négligeables comparés aux tailles importantes de paquet équivalentes à une ligne, un bloc ou une trame complète.

5.8 Conclusion

Dans ce chapitre, nous avons défini, proposé, implémenté et évalué une nouvelle architecture de routeur sur puce capable d'adapter dynamiquement son chemin de donnée interne sur plusieurs voies de communication en parallèle. Cette adaptation s'effectue de façon à réaliser de la manière la plus efficace possible les différents modes de fonctionnement requis entre les opérations dans une application donnée, dans le but de s'approcher

des performances d'une solution dédiée point-à-point, tout en ayant une flexibilité dans le chemin de données.

L'implémentation de la nouvelle proposition d'architecture de routeur démontre les performances en temps d'adaptation, la simplicité des communication des commandes et la consommation en surface raisonnable dans le contexte d'une solution utilisant des unités de calcul à grain-moyen.

En considérant que ces paquets de données peuvent atteindre des tailles importantes de l'ordre d'une trame image complète et que l'architecture adaptable que nous visons est capable de changer dynamiquement d'applications, notre système se confronte à un verrou important concernant l'aspect mémorisation des trames. La mémorisation de trame est d'autant plus importante dans le cadre d'application de vision exploitant l'aspect temporel. Nous étudierons alors dans le chapitre suivant différentes stratégies de mémorisation et en particulier les méthodes de conception traditionnelles performantes mais statiques. Nous présenterons alors par la suite notre nouvelle proposition de mémorisation adaptée à notre réseau de communication dynamiquement adaptable.

Chapitre 6

Système de mémorisation dynamiquement adaptable

6.1 Introduction

Dans un système de vision embarquée, nous appelons généralement par le terme *Frame Buffer* [146], un espace de mémorisation dédié au stockage de différentes trames d'image. Ces trames peuvent provenir soit directement des capteurs, ou être le résultat d'opérations effectuées par des unités de calcul. Le *Frame Buffer* est un élément clef des architectures de vision embarquée [7] et son utilisation est incontournable pour la synchronisation de différents flux de données à des cadencements différents [147].

Il peut être requis tout au long d'une chaîne de traitement aussi bien en amont pour l'acquisition des images qu'en fin de chaîne pour respecter les cadencements imposés par des normes d'affichage comme celles spécifiées par le VESA [148]. Dans le cadre d'une implémentation des applications de vision, la présence d'un *Frame Buffer* est impérative pour des opérateurs temporels nécessitant différentes trames d'image à des indices temporels différents.

Nous exposons dans ce chapitre la mise en oeuvre d'un nouveau système mémoire dynamiquement adaptable. Ce système est intégré à notre réseau de communication par l'intermédiaire des noeuds maîtres, contrôlant les transactions principales des paquets de données pour être traités au travers des noeuds esclaves.

Après un rappel sur l'organisation traditionnelle des mémoires dans une architecture de calcul pour la vision, nous décrivons notre système de mémorisation basé sur une gestion dynamique de différents emplacements mémoires. Ces emplacements sont déterminés par des indicateurs que nous imposons pour identifier le positionnement physique d'une trame en mémoire. L'étude de l'intégration du système mémoire dédié aux trames, dans notre réseau, nous amène par la suite à la conception du routeur maître, capable d'effectuer des requêtes de lecture et d'écriture de trame d'image. Nous terminons ce chapitre par le prototypage du système mémoire et du routeur maître, avec une évaluation en surface et en performance temporelle, dans le cadre d'une implémentation sur un FPGA.

6.2 Intégration d'un système mémoire de trames avec le réseau

Dans notre réseau de communication, nous avons vu au chapitre 4 que les routeurs *maîtres* contrôlent les transactions principales des paquets de données. Ces routeurs s'échangent ainsi mutuellement des paquets de données correspondant à différentes images de capteurs différents. Ils représentent à la fois une *source* et une *destination* pour les flux de pixels.

L'adaptation unique du chemin de données n'est cependant pas suffisante pour réaliser efficacement une chaîne de traitement composée d'unités de calcul dont les cadencements peuvent être différents. Ainsi, malgré les possibilités de pipeliner les unités de calcul dans notre réseau, si les bufferisations internes ne suffisent plus pour tenir les cadences imposées, alors les performances en temps d'exécution des calculs s'en trouvent dégradées. Cette dégradation impacte ainsi la cadence trame en sortie du système.

Dans le but de garantir les performances et de permettre l'utilisation d'opérateurs temporels pour nos applications, nous devons intégrer au réseau une organisation mémoire spécifique permettant de stocker différentes trames image. La mise oeuvre d'une organisation mémoire a pour but de réduire les goulots d'étranglement de données dans le réseau et d'assurer la synchronisation des flux de données. Notre réseau, complété avec un système de mémorisation, est proposé en figure 6.1.

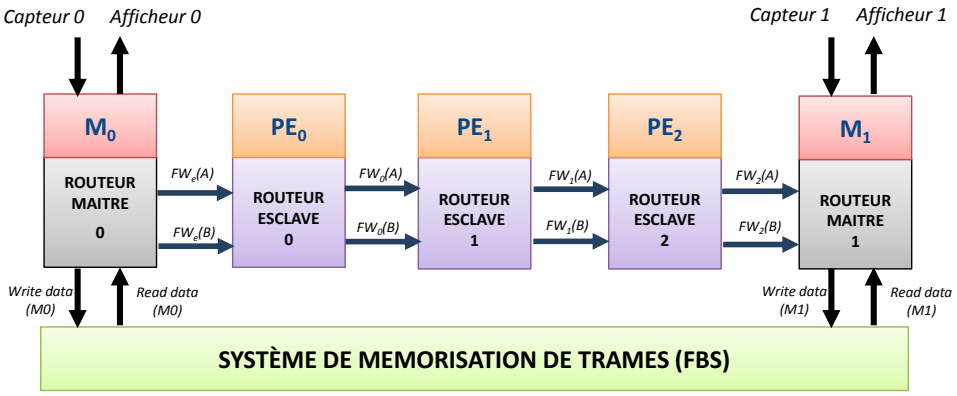


FIGURE 6.1: Modèle de réseau avec intégration d'un système de mémorisation de trames

Le système de mémorisation proposé, est intégré au réseau en le connectant à tous les routeurs maîtres. La mémoire dediée aux trames est alors partagée entre les routeurs *maîtres* et ils peuvent ainsi s'échanger des trames sans besoin de les communiquer par le biais du réseau. Cette solution permet ainsi de conserver le maximum de bande passante du réseau pour le traitement des paquets de données par les unités de calcul (PEs) raccordés aux routeurs *esclaves*.

Nous rappelons que les routeurs *maîtres* sont dotés d'interfaces d'entrée et de sortie, pour les flux pixéliques provenant des capteurs et vers les afficheurs. Les routeurs maîtres représentent ainsi les interfaces du réseau de communication avec l'extérieur.

Nous pouvons remarquer que ces interfaces avec l'extérieur auraient pu être considérées directement sur le système de mémorisation de trames. Ce choix impliquerait ainsi une simplification de la conception du routeur maître. Cependant, cette solution imposerait un chemin de données critique, comme celui présenté en figure 6.2(b), dépendant forcément d'accès en mémoire en entrée et en sortie. Ce choix augmenterait ainsi la latence de sortie. Elle ne permettrait donc pas d'établir un flot de données direct entre deux noeuds maîtres au travers des routeurs esclaves, comme illustré en figure 6.2(a).

(a) Sans mémorisation (b) Avec mémorisation

FIGURE 6.2: Types de chemin du flot de données

L'interfaçage des flux d'entrée et de sortie par l'intermédiaire des routeurs *maîtres*, présente ainsi l'avantage de pouvoir choisir entre les deux modes d'utilisation selon les

besoins de mémorisation.

6.2.1 Hiérarchie mémoire

Nous pouvons décomposer l'architecture d'un noeud *maître* en trois parties : une partie *contrôle*, une partie *mémorisation* et une partie réservée à l'*adaptation* du chemin de données et des accès en mémoire. Comme illustré en figure 6.1, chaque noeud maître possède un accès en lecture et écriture sur une mémoire partagée de grande capacité permettant de stocker différentes images qui sont véhiculées dans le réseau sous forme de paquets de données.

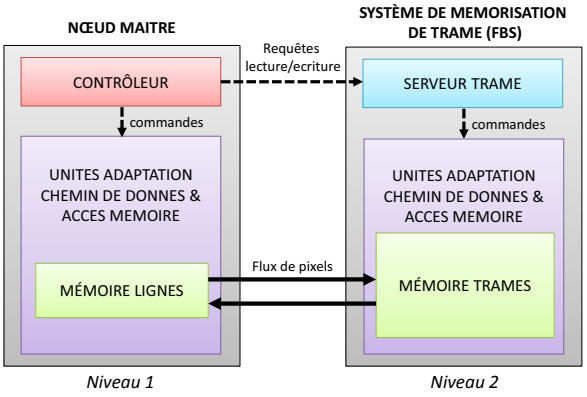


FIGURE 6.3: Hiérarchie mémoire : noeud maître et système mémoire de trames

Comme étudié au chapitre 2, dans le domaine de la vision, les unités de calculs manipulent généralement trois types d'entrée : le *pixel*, la *ligne* et la *trame*.

Dans le système complet réseau et mémoire que nous avons proposé, nous pouvons identifer deux niveaux de mémoire pour le noeud maître (Figure 6.3). Le premier niveau est une mémoire locale, à accès aléatoire, dont la capacité permet de stocker plusieurs lignes d'une trame image selon sa résolution. Le second niveau correspond aux mémoires de trames, à accès lignes, qui sont stockées dans le système mémoire partagé.

Le contrôleur, inclus dans le noeud maître, effectue des requêtes de lecture et d'écriture au système de mémorisation de trames. Chaque requête est réceptionnée par un serveur de trames central qui se charge de décoder les requêtes afin de récupérer les informations nécessaires à l'adressage de la mémoire. Suivant ces informations, il adapte le chemin de données et les accès mémoires pour diriger les flux de données entrants et sortants vers le routeur maître ayant effectué la requête. Le décodage et l'adaptation du système mémoire

introduisent une latence globale des différents accès qui doit être minimisée. Afin de masquer les latences d'adaptation d'accès en mémoire partagée mais aussi permettre des accès pixéliques spécifiques, les données trames peuvent être chargées partiellement par exemple dans les mémoires lignes contenues dans un noeud maître. Le tableau 6.1 résume les caractéristiques de chaque mémoire (type, maître, accès).

TABLE 6.1: Caractéristiques mémoires

Mémoire	Type	Maître(s)	Mode d'accès aux données
Lignes	Locale	1 noeud maître	Aléatoire
Trames	Globale Partagée	x noeuds maîtres	Lignes

Par rapport aux noeuds maîtres, les mémoires lignes sont locales et les mémoires de trames sont globalement partagées. Lorsque plusieurs lignes ont été chargées dans les mémoires locales, elles peuvent être accédées suivant différents modes (lecture successive, saut de pixels, formes spécifiques) prédéfinis dans les générateurs d'adresses contenus dans le noeud maître. Ces modes sont utiles dans le cas où les unités de calcul n'ont pas de mémoire locale et ne sont ainsi pas capable d'accéder de manière aléatoire sur une ligne de la trame.

6.3 Système mémoire en vision embarquée

6.3.1 Technologie mémoire volatile en vision embarquée

Dans le domaine de l'embarqué, les choix d'une technologie mémoire ainsi que l'organisation du système mémoire, est déterminante sur les performances en temps et en surface d'une architecture de calcul. Dans ce contexte, ces choix doivent s'effectuer de manière à réaliser un compromis en terme de performance (bande passante et latence d'accès), de consommation énergétique, d'encombrement et de coût de fabrication.

Une architecture de calcul, pour la vision embarquée, intègre généralement deux types de technologie mémoire volatile : la technologie DRAM [149] (*Dynamic RAM*) et la technologie SRAM [150] (*Static RAM*).

La distinction entre ces deux mémoires est faite dans un premier temps, par rapport à la fabrication physique d'un point mémoire. Dans le cas d'une DRAM, le point mémoire pour stocker un bit est réalisé à l'aide d'une capacité. La dénomination dynamique de

cette mémoire provient du fait qu'elle doit être rafraîchie périodiquement pour maintenir les niveaux de charge des capacités. Elle est opposée à la dénomination statique du point mémoire pour la technologie SRAM, qui est réalisé à l'aide de bascule, ne nécessitant pas de rafraîchissement.

Par rapport à la réalisation du point mémoire, nous pouvons déduire les impacts en terme de consommation et d'encombrement. En effet, la densité d'intégration est beaucoup plus importante dans le cas de la technologie DRAM que la technologie SRAM, dont la réalisation d'un point mémoire nécessite généralement 6 transistors pour maintenir l'information. Il en résulte de ce fait, que la consommation ainsi que le coût de fabrication d'une mémoire SRAM sont également plus importants que la DRAM.

Cependant, en terme de performance du point de vue de la latence d'accès aux données, la technologie SRAM est beaucoup plus efficace avec un accès qui est approximativement 4 fois plus rapide que la DRAM [151].

Avec un interfaçage de plus faible complexité que la DRAM, nous retiendrons que la technologie SRAM reste la solution privilégiée pour les applications critiques en terme de performance temporelle. Avec un coût de fabrication par bit plus faible, la DRAM est plus avantageuse pour des applications plus générales.

Il existe différentes déclinaisons de composants mémoires pour chaque technologie [152]. Pour la DRAM, nous pouvons citer par exemple, la SDR-DRAM (*Synchronous-DRAM*) correspondant à la première génération DRAM synchrone, la DDR-SDRAM (*Double Data Rate-SDRAM*) doublant la bande passante avec des transferts de données sur les deux fronts d'horloge ou encore la RLDRAM (*Reduced Latency DRAM*) qui est une DDR-SDRAM optimisée sur la latence d'accès aux données grâce à un bus d'adresse non multiplexé.

Pour la technologie SRAM, nous pouvons citer par exemple, la ZBT SRAM (*Zero Bus-Turnaround SRAM*) qui est une SRAM synchrone sans latence entre des opérations de lecture et d'écriture ou la QDR SRAM [153] (*Quad Data Rate SRAM*) possèdant deux ports de communication dont les données sont échantillonées sur les deux fronts d'horloge comme la DDR. La QDR permet ainsi un accès à 4 données en un seul cycle d'horloge avec des fréquences dépassant 200 MHz.

Des composants mémoires spécifiques sont également dédiés à l'embarqué avec une priorité sur la faible consommation énergétique. Nous pouvons citer la MDDR (*Mobile DDR*), aussi appelée LPDRAM [154] (*Low Power DRAM*), qui est une DDR SDRAM mais avec des tensions d'utilisation réduites et des techniques de rafraîchissement partiels. Citons également une mémoire basse consommation particulière comme la PSRAM [154] (*Pseudo SRAM*) qui est technologiquement une mémoire DRAM mais avec des interfaces proches de la SRAM.

Les composants FPGA intègrent également, directement dans la puce, des mémoires SRAM synchrones mais avec des capacités de stockage réduites par rapport à des composants dédiés. Les plus gros composants FPGA du marché peuvent intégrer des mémoires RAM dont la capacité de stockage est supérieure à la cinquantaine de Mbits[102]. Ces mémoires sont généralement utilisées pour implémenter des FIFOs ou mémoriser plusieurs lignes d'une image afin d'extraire un voisinage particulier, utile pour un simple convolueur par exemple, dans le cadre des applications de vision [98]. A défaut de ces mémoires, la mémorisation est effectuée au travers des LUTs et des registres du composant.

Dans le domaine de la vision embarquée, les deux technologies SRAM et DRAM sont souvent complémentaires dans une même architecture. La technologie SRAM est utilisée pour des applications nécessitant des temps d'accès très réduits. Nous pouvons citer par exemple un opérateur de stabilisation d'image qui nécessite d'accéder de façon aléatoire sur une partie de l'image dans le but de corriger dynamiquement le mouvement global estimé de l'image, en modifiant géométriquement l'image [155]. Cette technologie est donc particulièrement adaptée pour des accès pixéliques. Bien que les tailles des composants mémoires SRAM actuels sont suffisantes pour mémoriser plusieurs trames d'image de hautes résolutions, ceux-ci possèdent un coût relativement important par rapport à la DRAM. De manière générale, pour un simple objectif de stockage ou de temporisation, l'utilisation de composants DRAM est la solution la plus pertinente [146].

6.3.2 Méthode de stockage statique de trames

Dans un contexte de vision embarquée multi-applications, un système de mémorisation traditionnel peut être défini de manière à allouer statiquement des emplacements physiques pour chaque trame nécessaire à la réalisation d'une application.

La figure 6.4 illustre un exemple avec trois flux nécessitant d'être stockés respectivement en parallèle pour les applications $APP1$, $APP2$ et $APP3$. En supposant que la mémoire possède une bande passante suffisante à la fois en lecture et en écriture, un arbitrage est toujours requis pour les accès simultanés avec par exemple une méthode de type *round-robin*. Dans cet exemple, deux zones statiques sont allouées pour le flux d'entrée 1 (*Stream In 1*) dans la mémoire, afin de stocker celui-ci avec deux trames de retard au temps t et $t-1$. Ce double stockage de trame peut ainsi être utilisé soit pour des accès de type "ping-pong" (une trame après l'autre), ou soit pour pouvoir traiter deux images en parallèle avec des indices temporels différents.

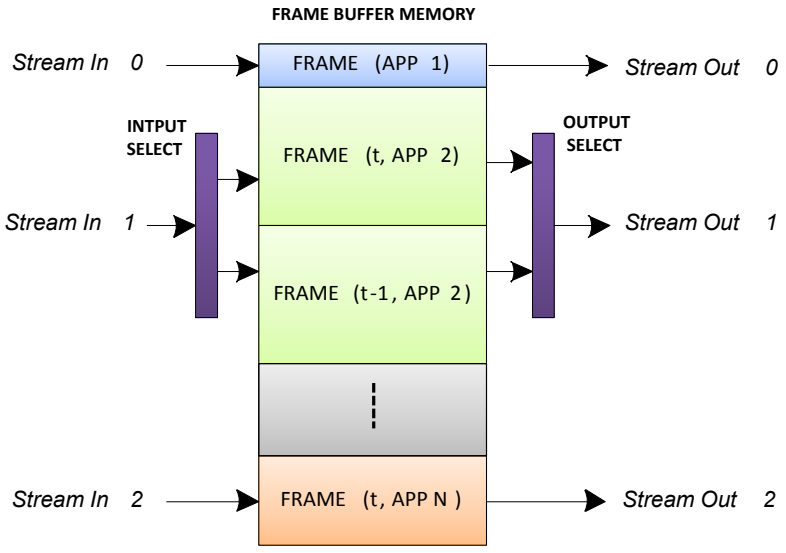


FIGURE 6.4: Partitionnement statique d'une mémoire dediée aux trames

Cette méthode statique simplifie grandement la gestion mémoire et garantie qu'aucun chevauchement de stockage de trame avec des conflits d'adresses physiques, ne peuvent intervenir pour des requêtes parallèles en écriture.

Le partitionnement de la mémoire est établi en fonction des différentes applications à implémenter. Chaque application possède des besoins spécifiques en ce qui concerne l'accès à un *type* de trame (monochrome, couleur) et au positionnement temporel de celle-ci. En effet, pour des applications nécessitant des opérateurs temporels, comme celui pour effectuer un débruitage temporel étudié au chapitre 2, il est indispensable d'avoir accès à des trames enregistrées à des délais précis.

Dans notre contexte où les applications peuvent varier au cours du temps, il devient alors difficile de partionner la mémoire de manière à prévoir tous les cas d'utilisation possibles. Par cette méthode, la variété d'application possible est alors limitée par la taille

de la mémoire dédiée au stockage des trames. Le partitionnement statique n'est donc pas la solution la plus efficace dans le cadre de notre modèle de réseau dynamiquement adaptable, car l'emplacement physique de l'adresse devrait, idéalement, être rédefini à chaque changement d'application.

Se trouve ainsi posé le problème de la gestion dynamique de l'espace mémoire, dans un contexte de réseau de communication, où les images circulent sous forme de paquets de données, dont les contenus sont identifiables à la lecture des en-têtes. Une nouvelle solution doit ainsi être proposée, avec une gestion des trames la plus simple possible, afin de ne pas complexifier le contrôle en lecture et écriture sur le système mémoire.

6.4 Méthode de stockage dynamique des trames

La nouvelle méthode de stockage des trames doit pouvoir tirer profit des informations contenues dans les attributs de chaque paquet de données, aussi bien pour l'enregistrement de l'image transportée, que pour son identification à la lecture en mémoire. Un des objectifs étant de supprimer toute association d'une application implémentée avec un emplacement physique et figé en mémoire.

Notre proposition, présentée dans cette section, repose sur un partitionnement de la mémoire dédiée au stockage de trames, en plusieurs emplacements adressables, appelés *slots*. Cet ensemble de slots est géré de manière dynamique pour leur allocation dans le cas d'une écriture et d'une désallocation éventuelle pour une lecture. L'adressage de ces slots s'effectue de manière indirecte en se basant sur des indicateurs définis à partir des attributs de l'image.

6.4.1 Définition d'un slot de trame en mémoire physique

Nous appelons *slot trame*, une trame image provenant d'un capteur n, traitée à un indice de temps ts fixé. Cet indice de temps ts représente la position temporelle relative d'une trame par rapport celle de même type en cours de traitement. Les différents *slots trame* d'image sont stockées dans une mémoire dédiée au stockage de trames appelée *mémoire de trames*. Une mémoire de trames peut ainsi contenir plusieurs *slots trame*.

Soient M et N la hauteur et la largeur respective d'un *slot trame*. Soit r la taille d'un pixel en nombre de bits. La taille du bloc mémoire S_F pour stocker ce slot trame sera :

$$S_F = M \times N \times r \tag{6.1}$$

Nous appelons *slots trame mémoire* l'emplacement physique du bloc mémoire de taille S_F alloué pour stocker un *slot trame*. Par exemple, la taille d'un *slot trame mémoire* pour une image monochrome HD 1920×1080 en 8 bits correspond environ à 2 Mbits.

Ainsi, soit S_{LM} la taille d'une *mémoire de trames*. Le nombre P de *slots trame mémoire* disponible dans la *mémoire de trames* sera :

$$P = \lfloor \frac{S_{LM}}{S_F} \rfloor \tag{6.2}$$

Par exemple, une *mémoire de trames* de taille 128 Mbits est capable de stocker théoriquement 64 *slots trame* monochrome HD 1080p en 8 bits.

6.4.2 Proposition d'un système de gestion adaptable de slots trame d'image

Le principe de notre système de mémorisation des trames, que nous appelons *Frame Buffer System* (FBS), est présenté en figure 6.5. Ce système est composé de deux parties : une partie mémorisation comportant une mémoire de capacité importante pouvant stocker plusieurs images différentes à différents indices temporels, et une partie contrôle se chargeant d'adapter les chemin de données en lecture et en écriture des différents flux de données entrants et sortants.

La partie contrôle se comporte comme une interface en entrée et en sortie de la partie mémorisation et représente une couche d'abstraction pour les requêtes d'accès à la mémoire.

Cette couche a pour objectif de masquer, au niveau système, les intéractions physiques avec la mémoire. Contrairement à la méthode statique précédente, ce système permet, en particulier, de décrire une application sans nécessité préalable de définir un espace

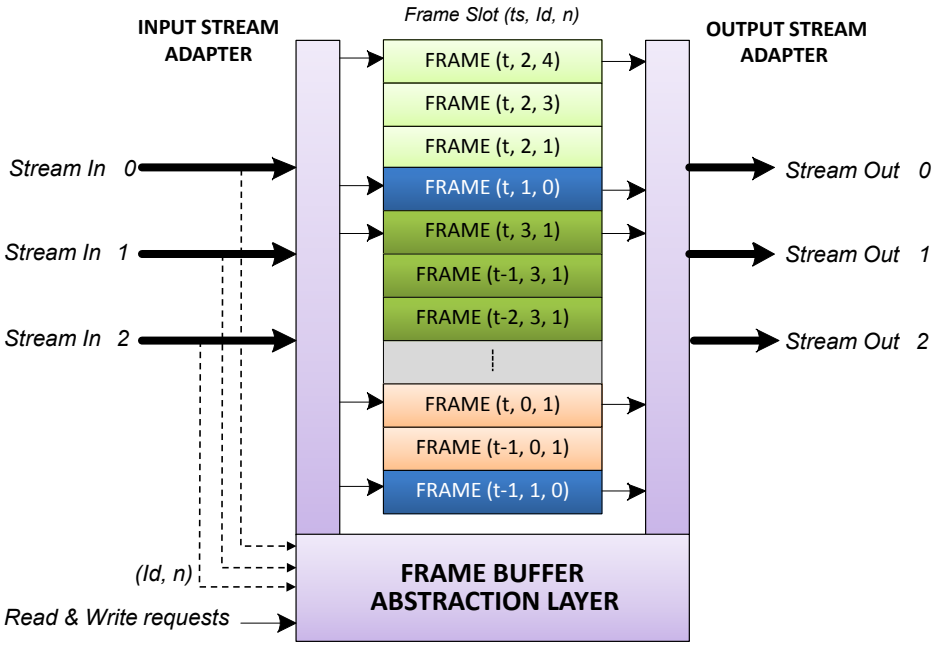


FIGURE 6.5: Partitionnement et gestion dynamique des slots trame

d'adressage dédié. Un besoin de stockage ou de lecture doit cependant être explicité de manière précise avec une description de l'image souhaitée. Cette description est basée sur l'identification d'une image sur trois indicateurs : le capteur utilisé id, l'indice temporel requis ts et la dernière opération appliquée n.

Comme illustré en figure 6.5, la mémoire est partitionnée en plusieurs slots identifiés par ces trois indicateurs.

Le premier indicateur, sur le capteur utilisé, permet ainsi d'informer sur la granularité et le type de pixel de l'image. Le second indicateur sur l'indice temporel détermine le positionnement temporel d'une trame dans le cas d'une mémorisation successive de plusieurs images issue d'un même capteur. Enfin, le dernier indicateur permet de déterminer la dernière opération effectuée. Cet indicateur informe également sur le positionnement de la trame dans une chaîne de traitement réalisant une application en cours d'exécution.

Dans l'exemple présenté dans la figure 6.5, trois canaux de communications, véhiculant des flux de pixels *Stream 0* à 2, provenant de capteurs différents, sont connectés sur le système de mémorisation. A chaque capteur est associé une application spécifique dont l'exécution nécessite le stockage des trames au cours du traitement. Pour le capteur 1, nous pouvons par exemple observer le stockage de deux slot trames contenant le résultat du calcul de l'opération 0 à deux indices temporels différents t et $t-1$, soit la trame courante et la trame précédente.

6.4.3 Avantages du système de mémorisation

Ce système de mémorisation, basé sur une gestion dynamique de multiples *slots trame*, présente plusieurs avantages.

Premièrement, l'originalité principale de ce système de mémorisation est que toutes les requêtes de lecture et d'écriture ne s'effectuent pas de manière explicite au travers d'adresses physiques en mémoire, mais par le biais des indicateurs identifiant une trame à stocker ou à écrire. Du point de vue des unités maîtres sur la mémoire, l'adressage physique en mémoire est alors complètement transparente.

Deuxièmement, ce système de mémorisation est très intéressant pour les applications utilisant des opérateurs temporels. Nous avons en effet observé que ces opérateurs étudiés au chapitre 2, peuvent nécessiter plusieurs trames de retard et une modification dynamique de cet opérateur pourrait avoir un impact sur l'indice temporel de la trame à traiter. Un opérateur peut tout simplement changer de repère temporel et ne souhaite plus se baser sur une mais deux trames de retard selon l'application.

Plus généralement, dans notre contexte de changement permanent d'application, ce système permet d'optimiser l'espace de stockage en désallouant les slots non utilisés. L'espace mémoire n'est alors pas sous-utilisé par des applications qui ne nécessitent que ponctuellement la mémoire.

Du point de vue évolutif, la méthode de gestion d'un nombre de slots définis, permet une réutilisation de la couche d'abstraction indépendamment de la taille de la mémoire de trame du système. Si la résolution des images augmente, seul le partionnement en mémoire physique est modifié et non le contrôle des allocations, car celui-ci peut toujours être défini pour travailler avec le même nombre de slots.

Nous pouvons remarquer que les indicateurs du type de pixel ainsi que la dernière opération effectuée, peuvent être informés grâce aux en-têtes des paquets de données du réseau. Ce système de mémorisation est donc particulièrement adapté pour notre réseau de communication avec toutes les informations requises directement associées avec la donnée image. Il ne reste alors qu'à la partie contrôle du système de mémorisation, de rajouter le dernier indicateur ts permettant d'assurer une cohérence temporelle entre les trames stockées. L'indexage des trames s'effectue ainsi de manière transparente au cours des différents accès en écriture en se basant sur le décodage des attributs d'une

trame fournis dans l'en-tête des paquets. A la lecture, la trame requise est explicitée par l'intermédiaire du triplet d'indicateurs sur le type de capteur id, l'indice temporel ts et la dernière opération effectuée n.

6.5 Conception du Stream Gate Manager

6.5.1 Architecture

Dans le cadre d'une proposition d'architecture, nous appelons *Stream Gate Managers* (SGM) les noeuds *maîtres* de notre réseau de communication.

En considérant les intéractions d'un SGM au sein du réseau, nous pouvons distinguer trois types de ports de données : les ports pour les canaux de communication du réseau, les ports d'entrée-sortie vers l'extérieur et les ports d'écriture-lecture vers le FBS.

Soit K_c le nombre de canaux de communication connectés au SGM, K_{ext} le nombre de ports d'entrée externes et K_{fbs} celui pour les ports de données en entrée avec la mémoire partagée. Le nombre K maximum de flux de données à gérer par un SGM, sera :

$$K = K_c + K_{ext} + K_{fbs} \tag{6.3}$$

Une vue globale de l'architecture d'un SGM associé à quatre canaux de communication, est donnée en figure 6.6.

Sur la figure 6.6, nous pouvons observer la disposition en entrée et en sortie des canaux de données 1 à 4 directement connectés au réseau ($K_c = 4$). Dans cette architecture, chaque SGM dispose d'un seul port de communication en entrée provenant du FBS ($K_{fbs} = 1$) et d'un seul port d'entrée provenant de l'extérieur ($K_{ext} = 1$). Dans cette configuration, le SGM doit pouvoir ainsi gérer en parallèle un maximum de $K = 6$ flux en entrée. Nous pouvons remarquer que le minimum de flux est de 4 avec $K_c = 2$, afin que les routeurs esclaves puissent réaliser les différents modes de fonctionnement.

Un SGM contient un système mémoire local dédiée au lignes, appelé *Line Memory System* (LMS), composé d'un ensemble de plusieurs mémoires nécessaires à la bufferisation des données provenant du réseau, de l'extérieur ou du FBS. Chaque SGM contient

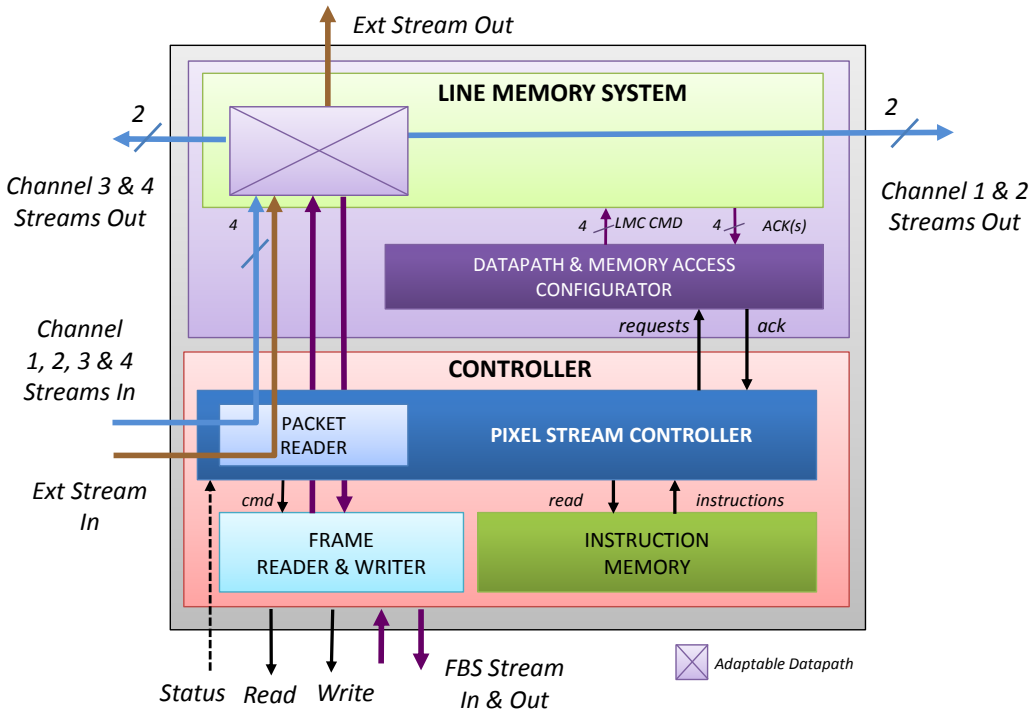


FIGURE 6.6: Architecture du Stream Gate Manager (SGM)

également une mémoire, appelée *Instruction Memory*, contenant le contexte de chaque application décrit par une suite d'instruction contenant les opérations à appliquer sur un type de flux particulier.

Les paquets de données entrants sont décodés dans un contrôleur global, appelé *Pixel Stream Controller*, dont le rôle est d'analyser les en-têtes des paquets et d'éditer si besoin cet en-tête. Cette édition permet de préscrire un nouvel ensemble d'opérations sur le groupe de données transporté par le paquet. Elle se termine lorsque le contrôleur atteint la dernière instruction et que toutes les opérations ont été appliquées sur la donnée. Lorsque qu'un paquet de donnée a été complètement traité et que le routeur maître correspond à celui de destination, alors les paquets de données sont ejectés à l'extérieur du réseau.

Suivant le décodage des en-têtes des paquets de données, le chemin de données du LMS est adapté dynamiquement en fonction de l'occupation des canaux de communications. L'état d'occupation des canaux est référencé dans l'unité *Datapath and Memory Access Configurator*. En fonction des instructions contenues dans l'en-tête du paquet de données, le *Pixel Stream Controller* peut effectuer des requêtes de lecture et d'écriture sur le système mémoire FBS.

6.5.2 Stockage du contexte

Chaque SGM contient une mémoire, appelée *Instruction Memory*, dédiée au stockage des contextes de plusieurs applications à exécuter en fonction des données en entrée.

Le contexte de l'application est décrit suivant un ensemble d'instructions. Nous rappelons qu'une instruction est composé de quatre champs : un champ [INST NUMBER] indiquant le groupe ou la ligne d'instruction, un champ [OP CODE] décrivant une opération, un champ [NB PASS] pour le nombre d'itérations requis pour l'opération et un dernier champ [TAG] permettant d'indiquer au DFR le mode d'exécution requis (séquentiel ou parallèle).

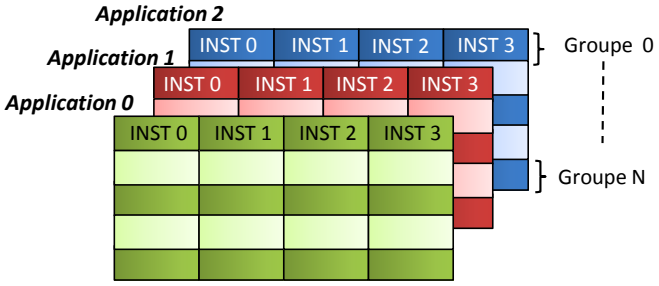


FIGURE 6.7: Stockage du contexte de chaque application en mémoire

Les instructions sont rangées en mémoire en plusieurs groupes dont la taille est égale au nombre d'instruction que peut contenir un en-tête dans le réseau. La taille des données en mémoire est ainsi définie en fonction de la taille d'un groupe. Par exemple, si nous considérons un en-tête capable de stocker un groupe de 4 instructions de taille 16 bits, alors la taille des données de la mémoire dédiée aux instructions sera de 64 bits. La figure 6.7 illustre un exemple de stockage de 3 applications en mémoire décrit par N groupes de 4 instructions.

Un contexte peut être chargé en mémoire à partir d'un nouveau paquet de données entrant dans un SGM. La figure 6.8 illustre un exemple de paquet transportant le contexte d'une application de la source d'image 0.

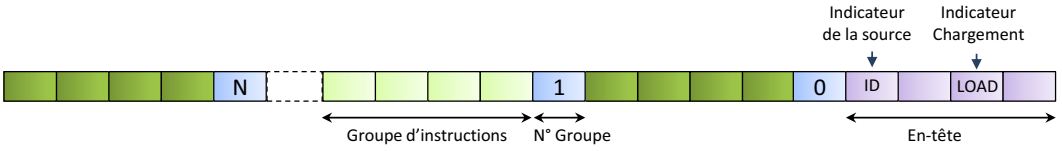


FIGURE 6.8: Données de chargement de contexte de l'application pour la source 0

Nous pouvons noter que les informations d'adresse et de données instruction sont multiplexées temporellement dans le paquet. Cette méthode autorise ainsi la modification partielle du contexte, sur des lignes d'instructions particulières, pour une application sans devoir effectuer une recharge complète. Le temps d'adaptation d'une application associée à une source d'image, est alors proportionnelle à l'importance des modifications.

Le paquet de chargement de contexte, doit cependant être transmis à tous les SGMs du réseau, afin de mettre à jour toutes les mémoires concernées.

6.5.3 Implémentation de la mémoire locale du Stream Gate Manager

6.5.3.1 Architecture du Line Memory System

L'architecture du LMS est présentée en figure 6.9 permettant d'adapter le chemin de données pour six flux en entrée ($K = 6$).

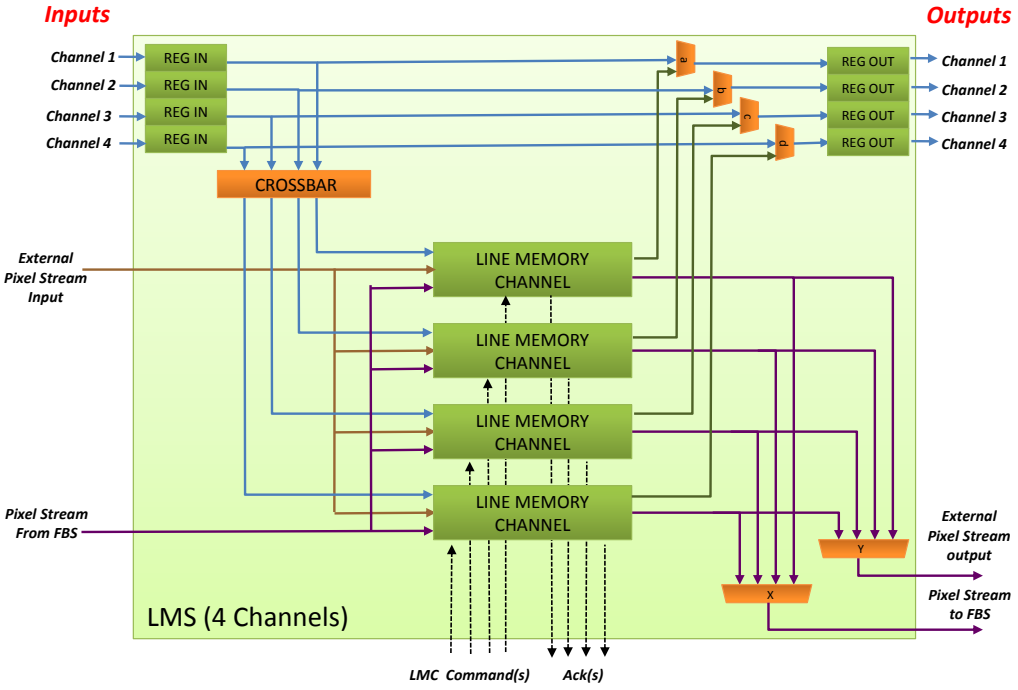


FIGURE 6.9: Architecture du LMS avec $K_c = 4$, $K_{ext} = 1$ et $K_{fbs} = 1$

L'adaptation du chemin de données est réalisée à l'aide de deux multiplexeurs à deux entrées, un crossbar à 4 entrées et d'un dernier multiplexeur à 4 entrées. Un LMS contient des sous-unités pour la mémorisation locale, appelées *Line Memory Channel* (LMC). Comme montré en figure 6.9, il y a autant de LMC que de canaux de communication. Un paquet de données de données entrant peut être mémorisé temporairement dans

une LMC afin de changer de canal de communication ou de préparer une écriture en mémoire globale. Le paquet peut également traverser complètement le SGM sans besoin de mémorisation si il n'en a pas l'utilité. Nous pouvons citer par exemple le cas d'un paquet entrant, non traité complètement par les routeurs esclaves, ou encore le cas où le SGM n'est pas celui de destination pour le paquet. Cette possibilité d'adaptation du chemin de donnée dans le SGM est importante afin de minimiser la latence de traversée d'un paquet dans un SGM mais aussi optimiser l'utilisation des LMC.

6.5.3.2 Architecture du Line Memory Channel

L'architecture du LMC est présentée en figure 6.10.

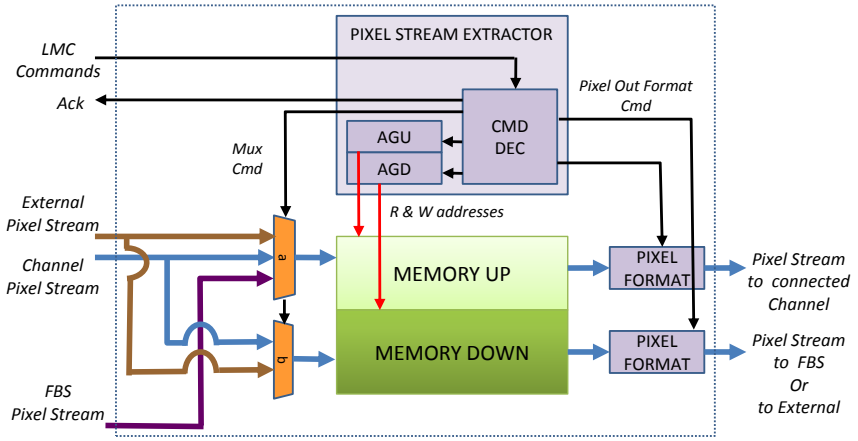


FIGURE 6.10: Architecture du Line Memory Channel

Chaque LMC est associée à un port de communication dans le réseau et il peut accepter des données provenant du réseau de communication, de l'extérieur ou de la mémoire globale FBS. Toute donnée stockée dans une LMC peut être envoyée soit vers le FBS, soit vers l'extérieur ou soit vers le réseau uniquement via le port de communication auquel il est associé. Le choix du LMC est donc déterminant pour la sortie des données. Comme décrit par la figure 6.10, un LMC est composée de deux mémoires appelées *Memory Up* et *Memory Down*. La première est réservée pour des données à transmettre vers le réseau de communication et la deuxième est utilisée pour les données vers le FBS ou l'extérieur. Ainsi, un paquet de donnée peut être chargé dans tout LMC pour un besoin d'écriture des données en FBS ou de transmission en sortie du système.

L'accès à ces deux mémoires ainsi que leur mode de lecture sont gérés par une unité de commande locale au LMC, appelée *Pixel Stream Extractor*. Celle-ci se charge de décoder

les requêtes d'adaptation provenant du contrôleur global du SGM. Elle aiguille ainsi les paquets de données sur les mémoires en configurant les multiplexeurs a et b, suivant la sortie désirée.

Les générateurs d'adresse (AGU et AGD) de lecture et d'écriture, pour les deux mémoires, peuvent être configurés pour charger linéairement en mémoire des données et les relire suivant des modes d'accès spécifiques. Les générateurs disposent de modes pré-définis pour accéder les pixels linéairement ou de manière aléatoire sur une fenêtre particulière en préchargeant plusieurs lignes. Cette fenêtre est limitée par la taille des mémoires locales permettant de charger un nombre maximum de lignes suivant la résolution de l'image. Nous pouvons par exemple citer un mode utile pour décimer une image par deux sans interpolation, en sautant un pixel sur deux, ou encore un mode permettant de recaler une image suivant un offset particulier sur l'axe horizontal et vertical pour l'affichage.

Suivant le mode appliqué et la taille en bit des données stockées en mémoire, il est nécessaire de formater les données sur la taille des bus de données du réseau de communication. Considérons en exemple une image, dont la granularité pixel est de 8 bits, stockée dans une mémoire locale dont la largeur des données est de 32 bits, qu'on souhaite accéder de manière à diviser la résolution par deux. En considérant un bus de données de taille 32 bits dans le réseau, le formatage consistera alors à appliquer un masque sur les octets non valides et à concaténer les octets retenus. Dans ce cas précis, deux accès à la mémoire seront ainsi nécessaire pour réaliser un flit de données sur 32 bits.

6.5.3.3 Contrôle des adaptations en chemin de données et des accès mémoires

Un LMS possède deux niveaux de sélection du chemin de données pour un paquet de données entrant. Le premier niveau correspond à la sélection du LMC, qui détermine le port de sortie du SGM dans le cas d'une retransmission du paquet de données sur le réseau. Le second niveau consiste à décider de la mémoire locale *Up* ou *Down* suivant la direction du paquet qui peut être stocké en mémoire, envoyés à l'extérieur du système ou retransmis sur le réseau.

Le chemin de donnée du paquet dans le LMS est décidé selon les cinq types de requêtes suivantes, provenant de l'unité de contrôle du SGM :

- FW(D) : Retransmission directe du paquet vers le port de sortie dans la même direction que le port d'entrée
- FW(C) : Retransmission avec un changement de direction
- READ : Requête de lecture d'un slot trame dans le FBS
- STORE : Stockage du contenu du paquet dans un slot du FBS
- OUT : Ejecter le paquet de données en sortie du système

Lorsque le paquet de donnée doit rester dans le réseau de communication, il est soit transmis vers le port de sortie qui est dans la même direction que celle du port d'entrée, ou soit redirigés vers un autre port pour changer de direction par exemple. Le premier cas correspond au mode $FW(D)$ et le second au mode $FW(C)$. Un exemple de ces modes est présenté en figure 6.11 avec le paquet entrant au canal 1 en mode $FW(D)$ et un paquet au canal 3 en mode $FW(C)$ qui est redirigé vers le canal 4 en sortie.

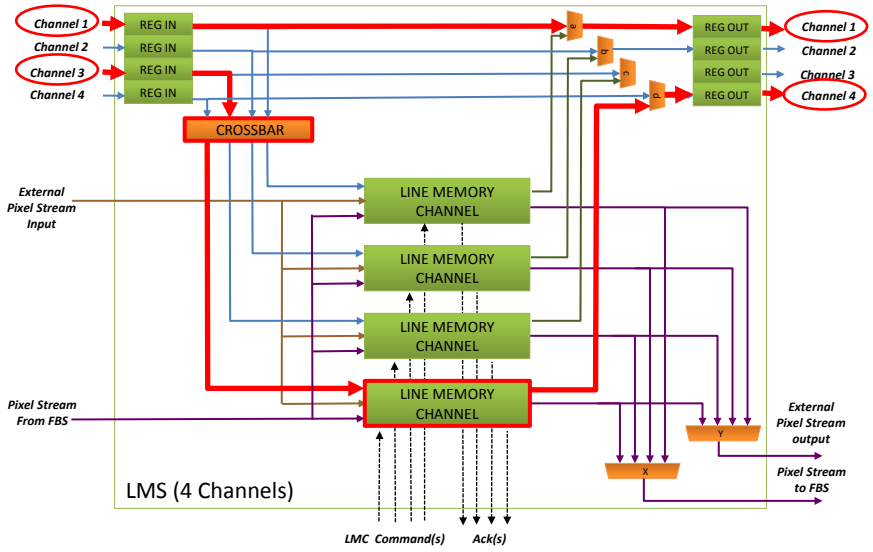


FIGURE 6.11: Mode FW(D) et(C)

Suivant les instructions contenues dans l'en-tête du paquet de données, il peut être requis de relire une trame stockée dans la mémoire globale. Ce mode READ, illustré en figure 6.12, introduit un nouveau flux de données pouvant générer un nouveau paquet dans n'importe quel canal du SGM. Le canal sélectionné est, par défaut, le premier disponible pour transmettre les données. Nous pouvons remarquer que plusieurs canaux de sortie peuvent être sélectionnés en même temps si on souhaite implémenter plusieurs applications différentes avec la même donnée d'entrée. La figure 6.12 illustre un exemple

pour la lecture d'un slot trame générant deux paquets : le premier vers le canal 1 et le second vers le canal 2.

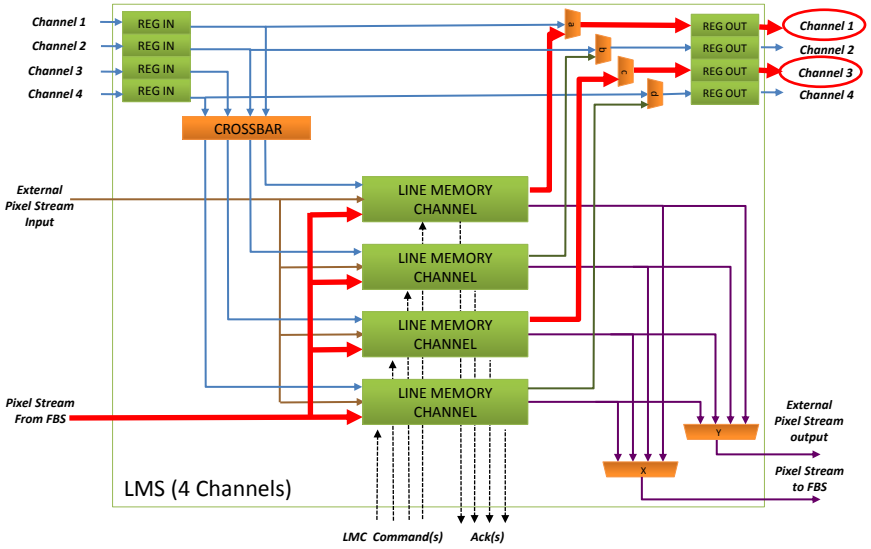


FIGURE 6.12: Mode READ

A l'inverse du mode précédent, le mode STORE est utilisé pour stocker les données d'un paquet entrant en mémoire. La figure 6.13 illustre un exemple avec un paquet entrant dans le canal 3, dont les données sont chargées dans le LMC du canal 3 pour le transférer en mémoire globale.

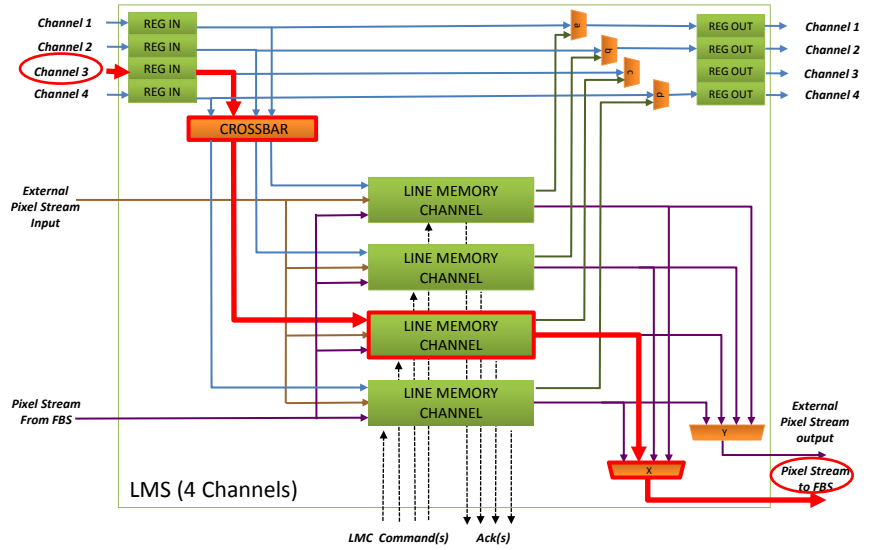


FIGURE 6.13: Mode STORE

L'adaptation du chemin de données du mode OUT est identique au mode STORE, à l'exception du port de sortie. La figure 6.14 illustre un exemple avec le même paquet dans le canal 3 mais destinés à être transmis à l'extérieur du système.

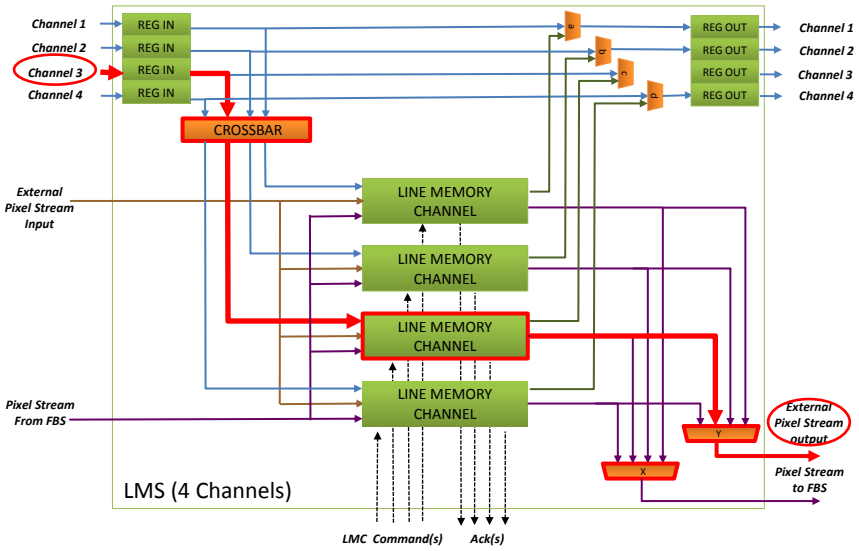


FIGURE 6.14: Mode OUT

Chaque type de requête mobilise un certain de nombre de ressources pour être appliqué. La table 6.2 résume le coût en nombre de mémoires et de ports pour chaque requête. L'adaptation du chemin de données est choisie en fonction de l'état d'occupation des ressources et du coût de la requête désirée.

TABLE 6.2: Utilisation des ressources par type de requête d'adaptation

	Mémoire		Ports du SGM			
	Up	Down	Réseau	Externe	Entrée FBS	Sortie FBS
FW(D)	0	0	1	0	0	0
FW(C)	1	0	1	0	0	0
READ	1	0	0	0	1	0
STORE	0	1	0	0	0	1
OUT	0	1	0	1	0	0

Nous pouvons remarquer que l'architecture proposée pour le LMS privilégie les sorties en écriture et en extérieur, dans le but de conserver le maximum de canaux et de mémoire lignes pour les besoins de traitement.

6.6 Conception du Frame Buffer System

Une proposition d'architecture du *Frame Buffer System*, reprenant le principe exposé en section 6.4, est présenté en figure 6.15 avec un banc composé de quatre mémoires dédiées aux trames d'image.

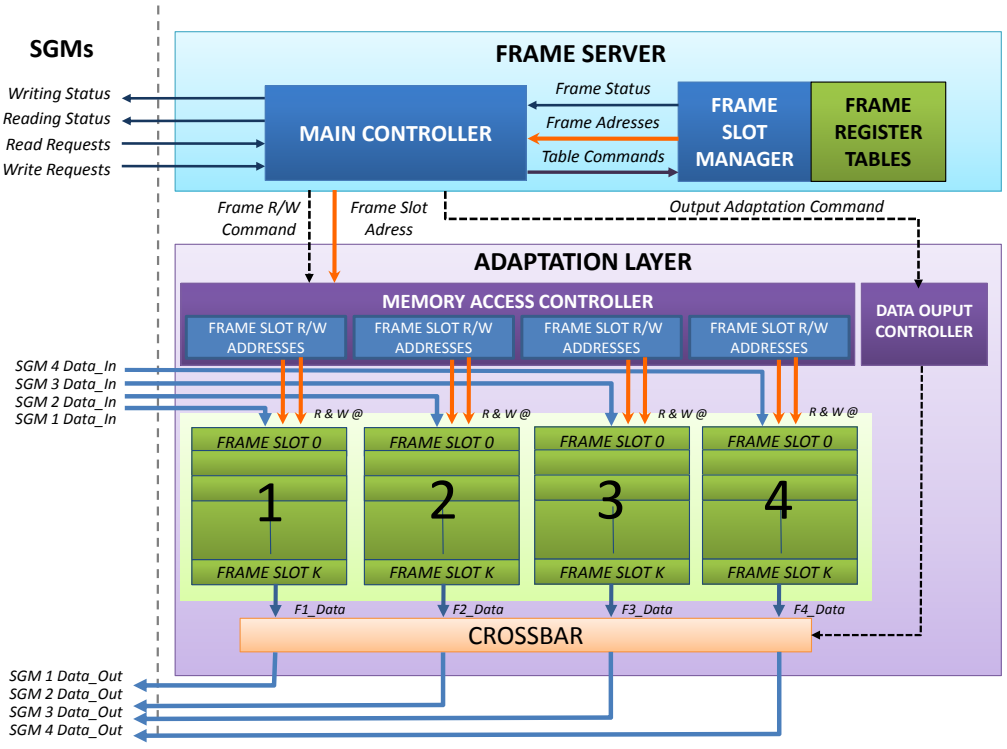


FIGURE 6.15: Architecture du Frame Buffer System avec 4 SGMs

Ce système est composé d'un contrôleur principal, ayant un rôle de serveur de trames (*Frame Server*), qui reçoit des requêtes de lectures et d'écriture, arbitre ces requêtes et commande les unités d'adaptation. Ces unités d'adaptation permettent de contrôler les accès mémoires (*Memory Access Controller*) et le chemin de donnée en sortie des mémoires de trames (*Data Ouput Controller*). Chaque mémoire de trames est découpée en plusieurs slots selon une division prédéfinie et supervisée par une unité de gestion dédiée, appelée *Frame Slot Manager*. Les unités de contrôle mémoire sont constituées de plusieurs générateurs d'adresses pour accéder à différentes parties de mémoire et l'unité d'adaptation du chemin de donnée en lecture, est constitué d'un crossbar pour atteindre les différents maîtres effectuant des requêtes sur la mémoire.

L'architecture optimale est réalisée avec autant de mémoires de trames que de noeuds maîtres afin de minimiser la latence en lecture. Cette disposition permet à chaque maître d'accéder en écriture de manière exclusive à sa propre mémoire de trames avec le maximum de bande passante. Elle évite ainsi un étage d'arbitrage supplémentaire en écriture, pénalisant la latence des accès en mémoire. Néanmoins, chaque maître est capable de lire le contenu de toutes les autres mémoires de trames qui sont partagées entre tous les maîtres exclusivement en lecture. Les requêtes d'écritures sont ainsi plus privilégiées

que celles de lecture, ce qui permet ainsi de minimiser le blocage des paquets de données dans le réseau.

6.6.1 Tables d'enregistrement des trames

Nous associons, à chaque slot trame, des attributs spécifiques (indice temporel ts, la granularité du pixel id, et la dernière opération n). Ces attributs sont mémorisés dans des tables d'enregistrement des trames (*Frame Register Tables*). Ces tables contiennent les correspondances entre les attributs des slots trames et l'emplacement physique en mémoire.

Comme montré en figure 6.16, les tables d'enregistrement des trames sont constituées de quatre tables principales : une table d'indexation des slots trames (*Frame Slot Memory Table*), une table d'adressage en mémoire physique (*Physical Memory Adress Table*), une table de stockage des attributs associés à chaque slot (*Frame Slot Attribute Table*) et une table indexant l'état de chaque slot (*Free Slot Table*).

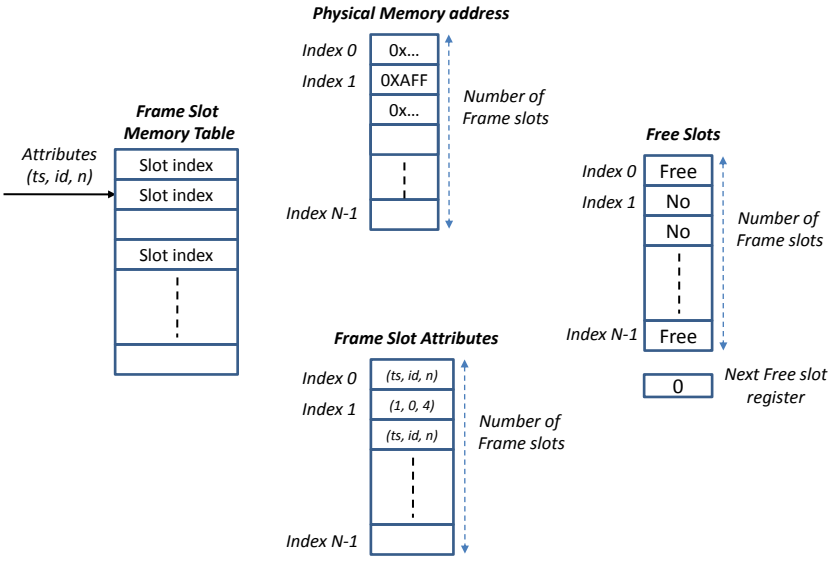


FIGURE 6.16: Tables d'enregistrement des trames

La *Frame Slot Memory Table* réalise la correspondance directe entre les attributs et le numéro de slot associé contenant l'information de la trame correspondante. La *Physical Memory Adress Table* contient les liens entre le numéro de slot et son emplacement physique en mémoire. La *Frame Slot Attribute Table* stocke tous les attributs (ts, id, n) associés à chaque slot et la *Free Slot Table* permet de surveiller l'état occupé ou libre de chaque slot. Nous pouvons également noter la présence d'un registre additionnel (*Next*

Free Slot Register) couplé avec cette dernière table et contenant le dernier slot trame disponible. Ce registre permet ainsi d'accélérer la recherche d'un slot libre dans le cas de requête en écriture.

Nous pouvons remarquer que la taille des tables ne dépendent principalement que du nombre de slots à gérer et non de la taille de la mémoire de trame. Il est tout à fait possible de ne travailler qu'avec un nombre et une taille définie de slots.

Selon la définition de la taille d'un slot, cette méthode de gestion permet de traiter des *blocs* d'une trame d'image séquentiellement ou en parallèle. Si effectivement une trame d'image est plus grande que la taille définie pour un slot, alors cette trame occuperait plusieurs slots et serait ainsi naturellement divisée en plusieurs sous-blocs image de plus petite taille à traiter. Chaque sous-bloc image pouvant ainsi être identifié selon un indice temporel croissant. Par exemple, une image HD 1080p de taille 1920×1080 peut être considérée comme 16 slots trames de taille 480×270 à traiter.

Cette technique est intéressante pour des PEs dont la puissance de calcul est limitée en fonction de la résolution de l'image, et nécessitent de traiter séquentiellement plusieurs sous-blocs. Pour accélérer un traitement de sous-blocs d'une image en parallèle, une trame doit pouvoir s'enregistrer sur plusieurs mémoire de trame afin de paralléliser complètement les accès en lecture. Cette méthode nécessite ainsi une transmission d'un même paquet de donnée vers plusieurs maîtres destinataires.

6.6.2 Gestion multi-slots de trames d'image

Le serveur de trames, *Frame Server*, contient des machines à états décrivant des algorithmes d'adaptation en lecture (Algorithme 5) et en écriture (Algorithme 6) pour les différentes mémoires de trames. Il comporte également un espace mémoire pour stocker une file d'attente de requêtes dans le cas de demandes simultanées sur la même mémoire de trames.

Soient Rc_i, Wc_i and Cc_i les requêtes respectives de lecture, écriture et désallocation de slot vers le *Frame Server* par le maître `i`. Soit `j` le nombre de maîtres connectés sur le *Frame Server*, `p` le pixel en entrée et `q` la taille d'un burst de pixels correspondant à un groupe de pixels à lire ou à écrire de manière consécutive. Soient `ID` l'identificateur de la source capteur, `n` la dernière opération réalisée et `ts` l'indice temporel.

Algorithme 5 : Algorithme pour l'écriture d'un slot trame

Entrée :
- Wc_i : commande d'écriture du maître i ;

Sortie :
ID,n,ts mis à jour ;

Variables :
- ID : identificateur de la source capteur de l'image ;
- n : identificateur de l'opération ;
- ts : identificateur de l'indice temporel ;
- q : taille de burst d'écriture de pixels ;

Fonctions :
- DecodeParameters(Wc_i) : retourne les attributs ID, n and ts ;
- Clear(ID,n,ts) : libère un slot mémoire ;
- CheckFree() : retourne le premier slot trame disponible ;
- Write(x,p) : écrit le pixel p à l'adresse physique x ;
- UpdateTS(ID,i) : rafraîchit tous les indices temporels pour le capteur ID et la mémoire de trame associée au maître i ;

```
1  pour i = 0 à j faire
2      { ID, n, ts, q } ← DecodeParameters(Wc_i);
3      si Wc_i = Cc_i alors
4          Clear(ID,n,ts)
5      sinon
6          x ← CheckFree();
7          tant que q > 0 faire
8              Write(x,p);
9              x ← x +1;
10             q ← q −1;
11         fin
12         UpdateTS(ID,i);
13     fin
14 fin
```

Pour la procédure d'écriture d'une trame, décrite par l'algorithme 5, le *Frame Server* doit dans un premier temps, décoder les attributs `ID`, `ts` et `n` correspondants à la trame image requise avec un burst de taille `q` (fonction `DecodeParameters{}` en ligne 2). Il demande ainsi au *Frame Slot Manager* la disponibilité d'un slot trame libre. En se basant sur les attributs décodés, le *Frame Slot Manager* va pouvoir adresser les tables d'enregistrement de trames et retourner au *Frame Server* l'emplacement physique en mémoire du dernier slot trame disponible (fonction `CheckFree{}` en ligne 6). La figure 6.17 illustre un exemple dans le cas où le slot retourné correspond à `x`=0 avec les paramètres décodés `ts`=1, `ID`=0 et `n`=3.

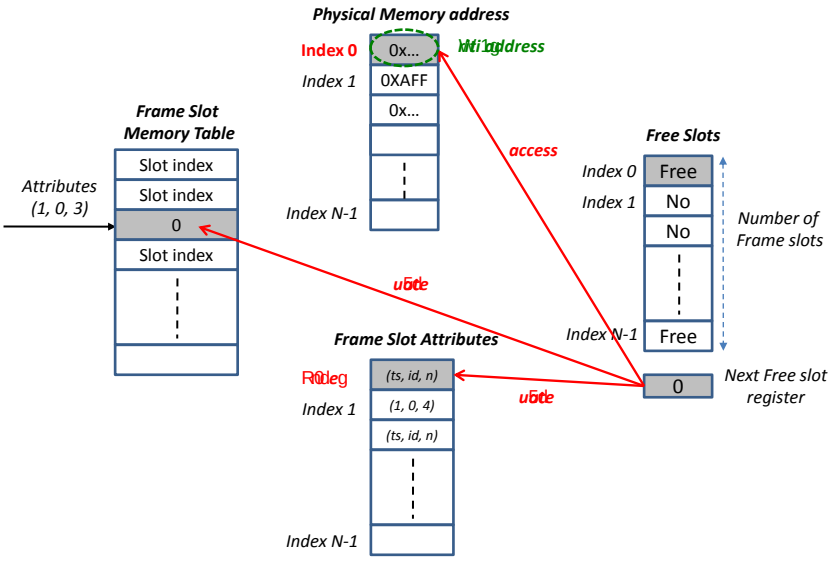


FIGURE 6.17: Ecriture d'une trame dans le slot 0

L'adresse mémoire physique de la trame est transmise au *Memory Access Controller*, configuré avec un burst de taille `q` pour initialiser les générateurs d'adresses en écriture pour la mémoire de trames (boucle `tant que` en ligne 7). Au cours de l'écriture, le *Frame Slot Manager* s'occupe de mettre à jour les tables d'enregistrement en liant le numéro de slot trame choisi avec les attributs décodés et l'emplacement physique (fonction `UpdateTS` en ligne 12).

La figure 6.18 illustre un exemple de lecture d'une trame provenant du capteur 0, traitée par l'opération 4 avec un retard d'une trame.

L'algorithme 6 décrit la procédure de la lecture d'un slot trame.

Le *Frame Server* doit décoder dans un premier temps les attributs pour obtenir l'emplacement physique (fonction `DecodeParameters{}` en ligne 3). Si la mémoire de trames

Algorithme 6 : Algorithme pour la lecture d'un slot trame

Entrée :
Commande de lecture;
- Rc_i: commande de lecture du maître i ;

Sortie :
ID,n,ts mis à jour;

Variables :
- ID: identificateur de la source capteur de l'image;
- n: identificateur de l'opération;
- ts: identificateur de l'indice temporel;
- q: taille de burst de lecture de pixels;

Fonctions :
- DecodeParameters(Wc_i) : retourne les attributs ID, n and ts ;
- Read(x) : lit le pixel p à l'adresse physique x ;
- BusyR(i) : vérifie si la mémoire de trame est occupé en lecture pour le maître i ;
- getFrame(ID,n,ts) : retrouve l'adresse physique du slot trame en mémoire;

```
1  pour i = 0 à j faire
2  |  tant que queue [ i ] n'est pas vide faire
3  |  |  { ID, n, ts, q } ← DecodeParameters(Rc_i);
4  |  |  si BusyR(i) alors
5  |  |  |  queue [ i ] ← Rc_i ;
6  |  |  fin
7  |  |  x ← getFrame(ID,n,ts);
8  |  |  tant que q > 0 faire
9  |  |  |  Read(x,p);
10 |  |  |  x ← x +1;
11 |  |  |  q ← q −1;
12 |  |  fin
13 |  fin
14 fin
```

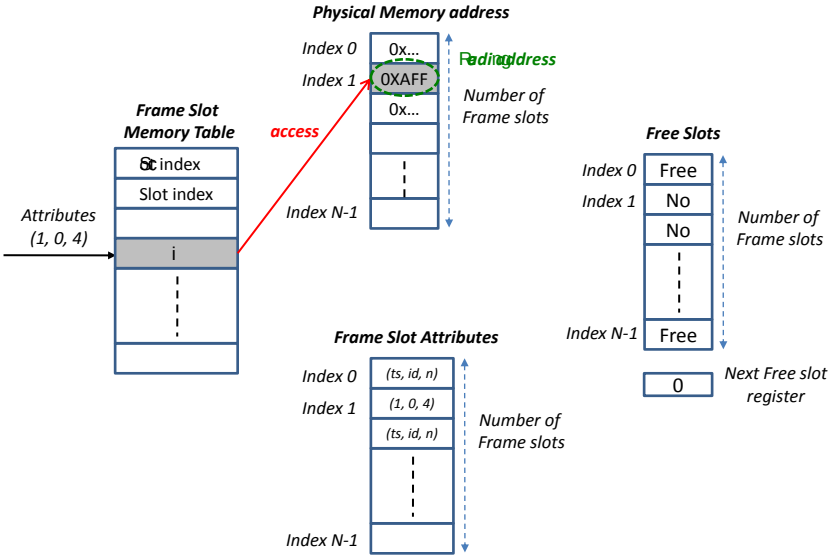


FIGURE 6.18: Lecture d'une trame enregistrée dans le slot 1

est occupée (fonction `BusyR{i}` en ligne 4), la requête de lecture est stockée dans une file d'attente `queue[i]` (ligne 5). Dans le cas contraire, le *Frame Server* commande le *Data Output Controller* afin d'adapter le chemin de donnée en sortie vers l'unité *maître* qui a effectuée la requête (fonction `getFrame{}` en ligne 7). Les générateurs d'adresse de lecture du *Memory Access Controller* sont initialisés à l'adresse mémoire physique obtenue par les tables avec un burst pixel de taille `q` (boucle `tant que` en ligne 8).

Dans l'exemple en figure 6.18, la lecture de la table *Frame Slot Memory* renvoit ainsi au *Frame Slot Manager* l'emplacement mémoire à l'adresse physique `0xAFF`.

6.7 Prototypage et Evaluation

Le *Stream Gate Manager* et le *Frame Buffer System* sont prototypés sur une cible Altera FPGA Stratix III EP3SL150. Pour évaluer ces implémentations, nous nous concentrons principalement sur le *coût* en surface (élements logiques) et en temps (latence) de la surcouche adaptative. Dans ce but, nous proposons dans ce prototype, de ne travailler uniquement qu'avec la mémoire SRAM embarquée sur la puce FPGA avec des tailles de slots réduits, pour valider notre système.

La tailles des slots n'étant pas déterminante, nous considérons le travail sur dix slots ($P = 10$) représentant des blocs d'images de taille $S_F = 10 \times 100$ octets. Nous considérons également que le système complet travaille à la même fréquence.

A l'exemple du prototypage du DFR, nous fixons une taille des canaux de communication à 32 bits.

6.7.1 Evaluation en surface du SGM

Les unités de contrôle du SGM, comme le *PixelStream Controller* et le *Datapath and Memory Access Configurator* sont implémentées sous forme de machines à états. Le *PixelStream Controller* contient une FIFO de 16 octets pouvant stocker des pixels durant l'adaptation.

Les mémoires locales *Up* et *Down* de chaque LMC sont implémentées sous forme de mémoires double ports de 32 bits de données, capable de stocker 2×100 octets soit 2 lignes de l'image. La Table 6.3 présente les résultats d'implémentation pour un SGM avec les paramètres $K_c = 4$, $K_{ext} = 1$ and $K_{fbs=1}$.

TABLE 6.3: Surface occupée par un SGM (Altera EP3SL150)

	ALUTs	Registres	Mémoire (bits)
Unités de contrôle	355	1495	960
LMS (4 LMCs)	520	768	16384
SGM (Total)	875	2263	17344
Occupation FPGA (%)	0.8	2	0.3

Nous pouvons remarquer que la taille des unités de contrôles est dépendantes de K_c, K_{ext} et K_{fbs}. Nous constatons que l'occupation d'un *Stream Gate Manager* dispose d'une occupation atteignant 2% en terme de logiques. Son occupation au niveau mémoire est dépendante du dimensionnement choisi des mémoires dans les LMCs.

6.7.2 Evaluation en latence du SGM

La table 6.4 montre la latence du SGM en nombre de cycles d'horloge d'un paquet de donnée entrant pour chaque mode d'adaptation : $FW(D)$, $FW(C)$, $READ$, $STORE$ et OUT.

La latence de traitement d'un paquet de donnée, dans un SGM, peut être dépendante de la latence d'adaptation du chemin de données et de l'accès aux mémoires locales.

TABLE 6.4: Latence (en cycles) pour un paquet entrant

Mode	LMS	Adaptation		Total
		Chemin de données	Mémoire	
FW(D)	2	2	0	4
FW(C)	$2 + \alpha$	6	β	$8 + \alpha + \beta$
READ	$2 + \alpha$	8	β	$10 + \alpha + \beta$
STORE	$2 + \alpha$	8	β	$10 + \alpha + \beta$
OUT	$2 + \alpha$	6	β	$8 + \alpha + \beta$

Le mode le plus rapide étant la transmission directe ($FW(D)$) en 2 cycles d'horloge du paquet de données avec le chemin de données qui n'est modifié qu'au premier niveau sans passer par un LMC.

Si le paquet doit être, au contraire, stocké dans une LMC avec les modes $FW(C)$, $READ$, $STORE$ et OUT, alors une latence supplémentaire de 4 cycles d'horloge est ajoutée pour communiquer avec le *Pixel Extractor* et attendre son acquittement. Dans le cas d'une lecture et d'une écriture, 2 cycles d'horloge supplémentaires sont requis pour initialiser les unités dédiées *Frame Reader* et *Writer*.

β est le temps des générateurs d'adresse pour calculer des accès spécifiques aux données. Dans notre implémentation, $\beta = 1$ pour un accès linéaire et $\beta = 3$ pour accéder à un bloc de pixel particulier.

Le temps α est la latence pour mémoriser les pixels dans une LMC suivant les mode d'accès. Par exemple, nous obtenons $\alpha = 50$ cycles d'horloge dans le cas d'un accès à une trame toutes les deux lignes de 100 pixels.

Nous pouvons remarquer que dans le cas d'une transmission directe $FW(D)$ n'étant pas réalisable, la latence sera dégradée à 6 cycles d'horloge pour changer de direction dans un LMC. Elle équivaut donc au mode $FW(C)$.

6.7.3 Evaluation en surface du FBS

Pour la mise en oeuvre du banc mémoire, nous utilisons quatre mémoires de trames implémentées avec des mémoires RAM embarquées double ports de taille 16384 octets afin de contenir $P = 10$ slots trames de résolution 10×100. Les unités de contrôle *Frame Server* et *Frame Slot Manager* sont également implémentées avec des machines à états. L'unité d'adaptation de l'adressage en mémoire est gérée par le *Memory Access*

controller contenant une machine à états couplée avec plusieurs compteurs 32 bits pour générer des adresses, au nombre de 8 dans notre cas.

La table 6.5 présente les résultats d'implémentation pour le FBS associé à une gestion de 10 slots trames. La fréquence maximum obtenue sur une cible Altera FPGA Stratix III EP3SL150 est de 138 MHz. Nous pouvons remarquer que la taille des slots trame n'impacte pas la taille des unités de contrôles ni les unités d'adaptation et les accès mémoires, si la taille des bus d'adresses et de données sont inchangées.

TABLE 6.5: FBS : Estimation de la surface (Altera EP3SL150)

	ALUTs	Registres	Memoire (bits)
Unités de contrôle	157	308	197632
Unités d'adaptation de chemin	264	0	0
Contrôleur accès mémoire	160	256	0
Mémoire de trames (4)	0	0	524288
Frame Buffer (Total)	581	564	721920
Occupation FPGA (%)	0.5	0.4	12.8

Nous pouvons constater que le coût en surface de la surcouche d'adaptation est tout à fait raisonnable pour la gestion d'une dizaine de slots trames.

6.7.4 Evaluation de la latence du FBS

La table 6.6 montre les résultats des latences (en cycles) mesurées pour le *Frame Slot Manager*. La valeur maximum mesurée pour T_{seek} est de 144 cycles d'horloge.

TABLE 6.6: Frame Slot Manager : Latences (en cycles) pour les requêtes de lecture et d'écriture

Requêtes	Latence
Lecture	T_{seek} + 4
Ecriture	T_{seek} + 5
Ecriture d'une nouvelle trame	3+T_{wrnf} =3+48

La table 6.7 montre que les latences obtenues pour le *Frame Server* dans les requêtes de lecture et d'écriture. Le *Frame Server* est en charge du décodage des requêtes pour commander le *Frame Slot Manager* et de l'adaptation du chemin de donnée après avoir obtenu les informations d'adressage physique des tables d'enregistrement.

δ_1 est le temps d'attente en lecture si la mémoire de trames est occupée. Il varie entre 0 et 250 cycles d'horloge pour lire un slot trame de taille 10×100 octets. δ_2 est le temps

d'attente pour l'écriture si le *Frame Slot Manager* est occupé à mettre à jour les tables d'enregistrement avec un maximum mesuré à 48 cycles d'horloge pour une gestion de 10 slots.

TABLE 6.7: Frame Server : Latences (en cycles) pour les requêtes de lecture et d'écriture

Requêtes	Décodage	Adaptation du chemin	Total
Lecture	1	$5+\delta_1$	$6+\delta_1$
Ecriture	1	$1+\delta_2$	$2+\delta_2$
Ecriture d'un nouvelle trame	1	$1+\delta_2$	$2+\delta_2$

De ces résultats mesurés pour les latences, nous pouvons conclure que le coût global en temps pour une adaptation varie entre 7 et 154 cycles d'horloge. Ces valeurs restent raisonnables en considérant la granularité importante de la taille d'une trame image et de la fréquence des accès pour les applications de vision. De manière évidente, plus la taille des slots trames est importante et plus les latences d'adaptation sont négligeables. Ces valeurs de latence permettent de dimensionner au mieux la taille des buffers en entrée et sortie de chaque maître afin de garantir un flot continu de pixels.

Nous pouvons noter que l'utilisation de mémoires externes à la place des mémoires internes sur puce, introduit un surcoût en latence qui est propre à la technologie et au composant parmi ceux présentés en début de ce chapitre. Un surcoût en surface est également à considérer en fonction du type de contrôleur mémoire utilisé. Notons cependant que dans le cas d'une implémentation FPGA, ce coût du contrôleur mémoire est moins impactant en ressources logiques si le composant intègre préalablement des contrôleurs physiques [156].

6.8 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle méthode de stockage des trames en mémoire. Cette méthode est basée sur un partitionnement de la mémoire en plusieurs emplacements physiques et un adressage de ces emplacements de manière indirecte par l'intermédiaire d'indicateurs caractérisant une trame dans une chaine de traitement.

Basée sur cette méthode, nous avons proposé une nouvelle architecture de système mémoire dédié au stockage des trames, appelée *Frame Buffer System* (FBS) et composée d'un banc mémoire associé à une surcouche d'abstraction adaptative.

L'étude de l'intégration de cette architecture au sein de notre proposition de réseau de communication nous amène à la conception du routeur maître, appelé *Stream Gate Manager* (SGM), ayant un accès exclusif sur le système de mémoire partagé. L'association entre les SGMs et le FBS établit une hiérarchie mémoire spécifique particulièrement adaptée à l'implémentation d'application de vision.

L'évaluation des architectures proposées pour le SGM et le FBS sur une cible FPGA, démontre les coûts en surface raisonnables permettant d'atteindre une plus grande souplesse dans la gestion des trames stockées. Les mesures en temps ont permis de confirmer que l'adaptation reste performante dans le cadre du réseau.

Le chapitre suivant sera consacré à la mise en oeuvre d'une architecture de réseau complète composée des différentes unités qui ont été étudiées. Nous étudierons et évaluerons dans ce chapitre l'implémentation d'applications de vision typiques dans un équipement développé au CE-CTP Sagem.

Chapitre 7

Validation Expérimentale

7.1 Introduction

Dans le but de valider expérimentalement notre réseau de communication, nous présentons, dans ce chapitre, une proposition d'architecture complète avec les unités de routage (SGM et DFR) et le système mémoire (FBS) proposés dans cette thèse. Cette architecture exploite différentes unités de calculs dédiées afin de réaliser des exemples d'applications concrètes en vision embarquée, développées au sein du CE-CTP de Sagem.

Cette architecture, que nous appelons *Multi Data Flow Ring* (MDFR), utilise une topologie du réseau en anneau, couplée à un système de mémoire de trames [19, 20].

La première partie de ce chapitre présente le choix topologique ainsi que la description de l'architecture. Les méthodes de chargement dynamique d'un contexte et de configuration des unités de calculs sont également décrites. Dans une seconde partie, nous présentons des résultats d'implémentation sur un FPGA avec une évaluation en surface et en temps dans le cadre d'une application concrète pour la restitution d'image bi-capteurs.

7.2 Architecture *Multi Data Flow Ring*

Dans notre réseau, la topologie globale est fixée par les routeurs maîtres (SGMs) qui sont reliés entre eux par un nombre variable de routeurs esclaves (DFRs).

Afin de valider notre proposition, nous choisissons une topologie en *anneau* permettant d'implémenter le plus simplement et efficacement possible des applications orientées flot de données, comme celles présentées au chapitre 2. Pour cette topologie, notre proposition de description des instructions dans les en-têtes se prêtent idéalement à des opérations récursives du fait de la présence d'un champ dédiée au nombre d'itérations nécessaires pour une opération donnée.

7.2.1 Présentation de l'architecture MDFR

Nous proposons ainsi l'architecture *Multi Data Flow Ring* (MDFR) présentée en figure 7.1.

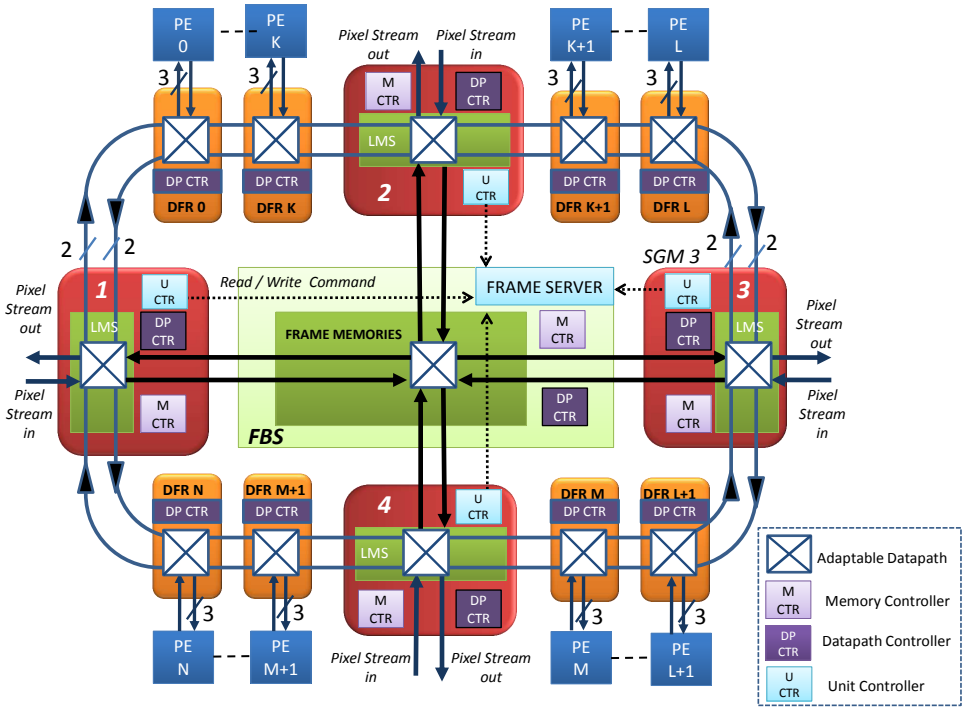


FIGURE 7.1: Architecture du *Multi Data Flow Ring*

Cette architecture est basée sur notre réseau de communication, organisé suivant une topologie en anneau de routeurs maitres (SGMs) et esclaves (DFRs). Les routeurs sont interconnectés suivant 4 voies de communications unidirectionnels : 2 voies orientées dans le sens des aiguilles d'une montre et les 2 autres voies dans le sens inverse. Cette configuration offre la possibilité au paquet de données de transiter dans les deux sens de circulation, ce qui lui permet ainsi d'atteindre un routeur maître de destination avec le

plus court chemin possible. Elle permet également au DFR d'utiliser le mode d'exécution parallèle dans les deux sens.

Les routeurs maîtres (SGMs) contiennent le contexte de l'application et contrôlent les mouvements principaux des flux de pixels. Nous rappelons que ce sont les seuls routeurs à être dotés d'interfaces d'entrée et de sortie de données.

Afin de pouvoir exploiter l'architecture, tous les nouveaux flux de pixels à traiter, provenant d'une source, sont préalablement paquetisés avec un en-tête. Cet en-tête contient au minimum l'attribut de l'image décrivant le type de source et une information d'adressage indiquant la position de l'afficheur de sortie souhaité dans le réseau. Cette position correspond à celle du SGM sur lequel l'afficheur est relié en sortie. Un SGM fournit alors à tout nouveau paquet de donnée entrant, un ensemble d'opération de traitement à appliquer, suivant le contexte de l'application associée à la source de pixel detectée.

Les SGMs s'échangent mutuellement des paquets de données sur un réseau linéaire de routeurs esclaves (DFRs) dont le rôle est d'adapter le chemin de données suivant les modes d'exécution des opérations, spécifiés dans l'en-tête du paquet de données. Le paquet de données ne peut sortir du système que lorsque toutes les opérations nécessaires ont été appliquées et qu'il a atteint le SGM de destination.

Le fonctionnement de l'architecture peut être compris par analogie à un réseau ferroviaire composé de gares principales (SGMs) et de stations intermédiaires (DFRs) séparant les gares. Un paquet de donnée peut s'apparenter à un train composé d'un chargement d'éléments à traiter. Chaque train entrant circule d'une gare de départ vers une gare de destination. Une feuille de route est fournie à la gare de départ contenant la gare de destination ainsi qu'un ensemble de traitement à appliquer à son chargement.

L'architecture MDFR peut être paramétrée avant synthèse suivant le nombre de SGMs et de DFRs. Afin de garantir le maximum de bande passante à la fois en traitement sur le réseau et en accès sur le système de mémoire FBS, nous limitons le nombre de SGMs à 4. Cette architecture autorise ainsi l'interfaçage de 4 sources d'images et de 4 afficheurs en sortie. Par ailleurs, avec les 4 voies de communications du système, les flux de pixels de chaque capteur peuvent ainsi transiter avec le maximum de bande de passante sur des lignes individuelles entre les SGMs.

Le nombre de DFRs dans le réseau est également proportionnel au nombre d'unités de calcul à intégrer dans le système. La latence maximum de chaque unité étant connu, la disposition des DFRs est effectuée de préférence en ne dépassant pas un seuil de latence maximum autorisé entre les SGMs. Ce seuil peut être défini suivant la distance entre une source image et l'afficheur de destination dans le MDFR.

7.2.2 Adaptation dynamique

Une application peut être modifiée dynamiquement dans le MDFR par l'injection de paquets spécifiques permettant de modifier le contexte, ce qui entraîne un changement dans le séquencement des opérations. Il est également possible de modifier directement le fonctionnement des unités de calcul par des paquets de configuration. Suivant ces changements, les routeurs s'adaptent automatiquement, en fonction du changement des instructions dans les en-têtes des paquets, avec les mécanismes présentés précédemment dans ce manuscrit au chapitre 5. De plus, dans le cas où la source de l'image venait à changer, résultant d'une modification de l'ID du paquet, le basculement d'application s'effectue automatiquement en chargeant de nouvelles instructions pré-chargées préalablement en mémoire du SGM.

7.2.2.1 Méthode de chargement du contexte

Dans le cas du MDFR, la topologie en anneau est particulièrement efficace pour une information dite de type *broadcast* à transmettre sur tout le réseau. Il suffit ainsi de définir pour le paquet le même SGM à la fois en source et en destination, puis de lui appliquer dans son en-tête les instructions résumées dans le tableau 7.1.

N° Instruction	Instructions
0	[1] [LOAD] [1] [/PAR]
1	[1] [OUT] [1] [/PAR]

TABLE 7.1: Instructions pour un paquet de chargement de contexte

La première instruction [LOAD] permet d'identifier le type de paquet pour le chargement de contexte et elle est traitée par tous les SGMs. Dès le passage du paquet dans le SGM d'entrée considéré comme source, le champ `NB PASS` de l'instruction est modifié à 0. Le

paquet parcourt ensuite l'intégralité de l'anneau avant de revenir au SGM de départ qui exécute l'instruction suivante [OUT] pour la sortie du paquet.

7.2.2.2 Configurations des unités de calcul

Les unités de calcul, intégrées dans le système, peuvent être configurées par différents paramètres, transmis dans le réseau sous forme de paquets de données spécifiques. Ces paramètres sont par exemple utilisés pour définir une zone d'intérêt dans l'image en transmettant les coordonnées des points utiles ou encore pour modifier un ensemble de coefficients de calcul.

Les instructions contenus dans le paquet sont similaires à celles du paquet de chargement de contexte, présentées précédemment. La seule différence réside dans l'identifiant permettant d'identifier l'opération à modifier. Si la position de l'opération n'est pas connue dans l'anneau, il reste possible de faire suivre à ce paquet le même trajet que celui du chargement de contexte, en faisant un tour complet dans l'anneau. Ce tour complet étant défini avec un SGM de départ et de destination identique.

Les paramètres des unités de calcul sont stockés dans des registres internes ou externes. Des registres externes sont utilisés pour configurer l'unité de calcul en parallèle. Les configurations de chaque registre sont multiplexés temporellement dans le paquet de données. La configuration en parallèle d'une unité de calcul s'effectue de manière transparente. Le démultiplexage des données est réalisé avec une structure dédiée faisant interface avec l'unité de calcul, comme présenté au chapitre 5.

En résumé, l'adaptation dynamique d'une application peut s'effectuer de différentes manières dans le MDFR : en basculant de contexte d'application en mémoire, en rechargeant complètement ou partiellement un contexte, et en modifiant localement une unité de calcul.

7.3 Implémentation et Evaluation

7.3.1 Exemple d'une application en vision embarquée

Pour valider notre architecture, nous allons implémenter une application typique d'un équipement de vision portable bi-capteurs développé au CE-CTP Sagem, permettant la visualisation d'une image sur un afficheur. L'objectif de ce prototype est d'évaluer le fonctionnement des différents mécanismes d'adaptation de notre architecture au niveau du chemin de données et des accès mémoires.

L'application que nous proposons, illustrée en figure 7.2, comporte différentes unités de calcul : un unité de sélection de région d'intérêt (ROI), une unité d'interpolation de l'image en Y (INTPOL Y), une unité d'interpolation de l'image en X (INTPOL X), une unité de changement colorimétrique (COLOR) et une unité d'incrusation d'image (INCRUST).

Dans cet exemple, nous supposons que les deux capteurs sont monochromatiques et que les données des trames en sortie sont paquetisées avec un en-tête permettant d'identifier la source.

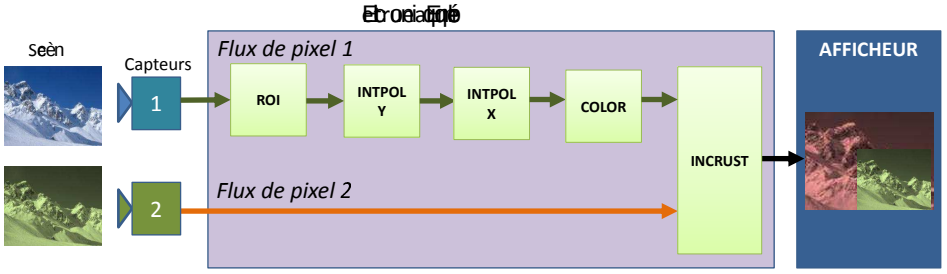


FIGURE 7.2: Exemple d'application en vision embarquée

L'unité ROI consiste à sélectionner un région d'intérêt dans l'image dont nous souhaitons avoir un grossissement. Cette région de l'image doit être par la suite stockée en mémoire puis relue pour effectuer sa transformation géométérique par interpolation. Cette région est alors interpolée suivant les axes Y puis X (INTPOL Y et X) permettant ainsi de réaliser un zoom numérique. Les niveaux de gris peuvent être ensuite modifiés par l'unité COLOR.

Cette application permet de fusionner deux images provenant de deux capteurs différents 1 et 2, avec le contexte que le capteur 1 dispose d'une résolution d'image inférieure à celle

de l'image du capteur 2. Un flux unique est alors produit en fonction des deux sources de pixels à l'aide d'une unité (INCRUST) capable de traiter deux flux en parallèle.

7.3.2 Adaptation architecturale pré-synthèse de l'architecture MDFR

Considérons que nous disposons de l'ensemble des PEs capable de réaliser l'application complète (ROI, INTPOL Y, INTPOLX, COLOR, INCRUST). A partir de cet ensemble de PEs à notre disposition, nous pouvons définir une architecture de type MDFR, capable d'implémenter l'exemple d'application présentée précédemment. Nous proposons ainsi une architecture composée de 4 SGMs, 5 DFRs et un FBS avec l'intégration des 5 opérateurs sur chaque DFR, comme illustré en figure 7.3.

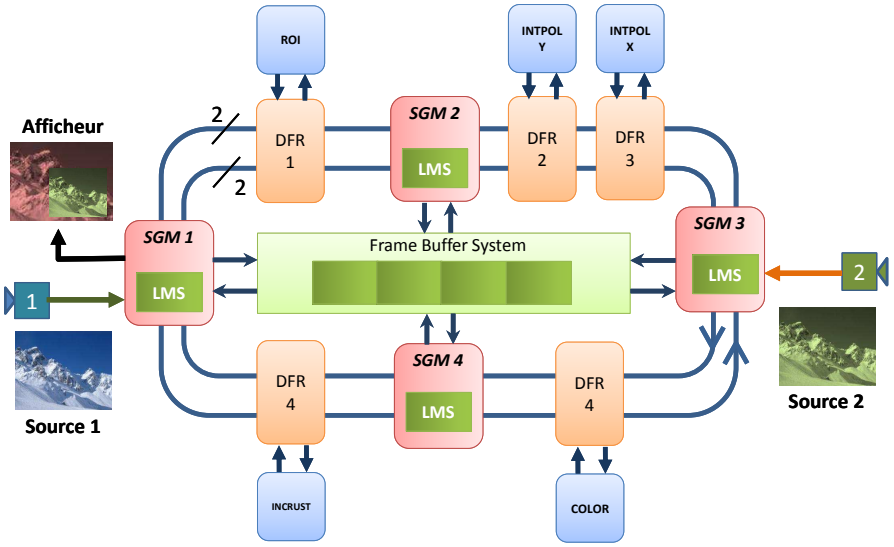


FIGURE 7.3: Paramétrage du MDFR avec 4 SGMs, 5 DFRs et 1 FBS

7.3.3 Implémentation de l'application sur l'architecture MDFR

Afin d'implémenter l'application, nous proposons la structure d'en-tête, résumée dans le tableau 7.2, dans le cadre de flits de données de 32 bits.

Chaque instruction, fixée à 16 bits, est codée de la manière présentée dans le tableau 7.3.

Nous associons à chaque source d'image, une application exprimée par une liste de lignes d'instruction. La taille d'une ligne d'instruction est égale à la taille totale des instructions

N° Flits (32 bits)	Contenu (taille en bits)
0	Header Start (0xFFFFFFFF) [31..0]
1	Nombre de pixels [31..0]
2	Instruction 0 [31..16], Instruction 1 [15..0]
3	Instruction 2 [31..16], Instruction 3 [15..0]
4	Plage réservée [31..13], Bit config [12], ID [11..8], Indice temporel [7..4], Noeud Maître Source [3..2], Noeud Maître Destination [1..0]
5	Header End (0xFFFFFFFF) [31..0]

TABLE 7.2: Détail du contenu de l'en-tête sur 6 flits de données 32 bits

Champ	Taille (bits)	Position	Information
INST NUMBER	4	[15..12]	Numéro de la ligne instruction
OPCODE	6	[11..6]	Type d'opération requise
NB PASS	4	[5..2]	Nombre de passe requis
TAG	2	[1..0]	Indication de traitement en parallèle

TABLE 7.3: Structure d'une instruction sur 16 bits

que peut contenir un en-tête de paquet de données. Cette taille est de 4×16 bits soit 64 bits dans notre exemple, pour 4 instructions.

Le codage de l'application associée à la source image 1, est indiqué par le tableau 7.4. Il est composé de deux lignes d'instructions. La première décrit l'utilisation de l'opérateur de sélection de la région d'intérêt de l'image (opération ROI) avec le stockage de cette région en mémoire (opération WRITE). L'opération d'écriture est suivie par celle de lecture (opération READ). Chaque instruction de lecture est suivie par les informations nécessaires pour accéder à la trame requise. Dans notre exemple, il s'agit de la trame précédente de la même source 1 dont la région a été selectionnée.

N° ligne	N° Flit	Instructions
1	0	[1] [ROI] [1] [/PAR]
	1	[1] [WRITE] [1] [/PAR]
	2	[1] [READ] [1] [/PAR]
	3	$[id = 1]$ $[ts = 1]$ $[op = ROI]$
2	0	[2] [INTPOL Y] [1] [/PAR]
	1	[2] [INTPOL X] [1] [/PAR]
	2	[2] [COLOR] [1] [/PAR]
	3	[2] [INCRUST] [1] [PAR]

TABLE 7.4: Instructions pour la source d'image 1

La seconde ligne d'instructions contient la succession d'opérations d'interpolation en Y puis en X (opérations INTPOL Y et X) afin d'effectuer la transformation géométrique, suivi d'une opération de colorisation. Cette dernière opération consiste à modifier une plage de niveaux de gris en un niveau spécifique. La ligne d'instruction se termine par l'utilisation d'un opérateur d'incrustation nécessitant de combiner en parallèle deux images.

La seconde image provient de la source d'image 2 dont le code instruction associé est présenté dans le tableau 7.5. Etant donné qu'il s'agit de la donnée à incruster, ses opérations ne consistent qu'à appliquer l'opérateur INCRUST puis à sortir les données (opération OUT) du réseau de communication en définissant le SGM 1 comme celui de destination. La taille total des instructions pour l'application sur ces deux sources d'image est ainsi de 192 bits.

N° ligne	N° Flit	Instructions
1	0	[2] [INCRUST] [1] [PAR]
	1	[1] [OUT] [1] [/PAR]

TABLE 7.5: Instructions pour la source d'image 2

A partir de ces instructions appliquées sur l'architecture MDFR définie, les chemins des paquets de données, provenant des sources d'image 1 et 2, sont illustrés en figure 7.4. Ils sont tracés en pointillés rouge pour la source 1 et marron pour la source 2. Pour les paquets provenant de la source 1, la première ligne d'instruction est exécutée entièrement entre le SGM 1 et 2.

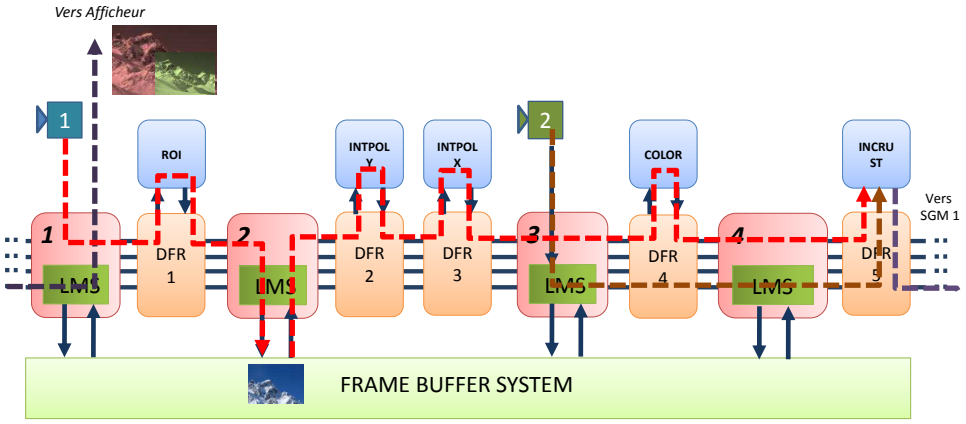


FIGURE 7.4: Implémentation de l'application sur le MDFR

Il est important de noter que tout paquet extérieur possède un en-tête permettant d'identifier la source d'image et sa destination. Notons également que les lignes d'instructions

en mémoire commencent à l'indice 1. En effet, deux cas d'en-tête sont possible pour exécuter le calcul sur la donnée en entrée : soit l'en-tête comporte déjà la première ligne d'instruction et le paquet peut déjà commencer à être traité ; ou soit l'en-tête contient par défaut l'instruction [0][X][0][/PAR] qui indique ainsi au SGM de passer à la ligne d'instruction 1 (la première ligne d'instruction).

7.3.4 Evaluation

Dans le cadre de notre évaluation, nous nous concentrons principalement sur le surcoût en latence d'adaptation et en surface de notre réseau de communication. Ainsi, nous travaillons essentiellement dans le cadre de ce prototype qu'avec des résolutions d'image de taille réduite ce qui permet ainsi de n'utiliser que la mémoire RAM embarquée dans un FPGA. En effet, étant donné que les unités de calcul dans notre exemple d'application peuvent être pipelinées, la résolution de l'image possède peu d'impact sur la latence de traitement entre la première donnée en entrée et celle de sortie de la dernière unité de calcul. La seule variation de cette latence peut être induite par les accès en mémoire de trames en lecture et écriture, selon la taille de l'image. Dans notre cas, celle-ci est variable selon la taille de la région d'intérêt.

Nous choisissons ainsi dans nos mesures de traiter une image de taille 64×32 dont la région d'intérêt de taille 8×8 est interpolée pour obtenir une résolution finale de 40×40.

Les mesures des latences sont présentées en tableau 7.6.

La latence de chargement d'un contexte sur l'anneau complet est mesuré à 820 ns (soit 82 cycles d'horloge). Le temps de chargement du contexte pour la source d'image 1 est alors de 88 cycles d'horloge pour un paquet transportant les données d'instructions sur 6 flits de 32 bits. Ainsi, suivant la résolution de l'image et la fréquence trame, il est possible de modifier dynamiquement le contexte entre deux trames.

Nous constatons que les latences d'adaptation les plus importantes pour le chemin complet, concernent les accès en mémoire FBS à la fois en écriture et en lecture. Pour cette application complète, le surcoût en latence, équivalent à une soixantaine de cycle d'horloge, correspond à 8 % de la latence totale du traitement. Ce surcoût est acceptable tout en sachant que nous ne travaillons qu'avec des résolutions d'image très réduites. De

Unités	Action	Latence (ns)	Latence (cycles)
SGM 0	routage DFR	100	10
DFR 0	calcul ROI	1570	157
SGM 1	routage FBS	120	12
FBS	écriture	4620	462
FBS	lecture	690	69
SGM 1	routage DFR et édition	70	7
DFR 1	calcul INTPOL Y	270	27
DFR 2	calcul INTPOL X	200	20
SGM 2	routage DFR	100	10
DFR 3	calcul COLOR	130	13
SGM 3	routage DFR	100	10
DFR 4	calcul INCRUST	140	14
SGM 0	routage extérieur	110	11
Latence du chargement contextes		820	82
Latence traitement total		8220	822
Latence d'adaptation chemin complet		660	66
% Latence adaptation chemin/traitement		8 %	

TABLE 7.6: Evaluation de la latence d'adaptation du réseau (fréquence 100 MHz)

manière évidente, avec une structure entièrement pipelinable, plus la taille des données du paquet est importante et plus le surcoût des latence d'adaptation devient faible.

Les résultats d'implémentation de l'architecture complète sont présentés dans le tableau 7.7.

Unités	Elements Logiques	Registres	Mémoire (bits)
4 SGMs	6040	9808	81920
5 DFRs	5885	9480	0
FBS	1082	770	376967
MDFR (total)	13007	20058	45887
PEs (total)	1356	1809	113152
% Occupation FPGA PEs	2	2	2
% Occupation FPGA MDFR	11	18	9

TABLE 7.7: Evaluation en surface du MDFR avec 4 SGMs (Altera EP3SL150)

Dans ce tableau, nous pouvons constater que l'architecture MDFR atteint une surface consommée de l'ordre de 10 % d'un composant FPGA comportant environ 150 000 élements logiques (l'élement logique est l'unité de base d'un composant FPGA Altera). Cette consommation est fortement liée à la demande en ressources registres du fait du nombre de voies (4), de la taille du bus de données (32 bits) et de la taille d'un en-tête de paquet (équivalent à 192 bits).

Cette surface peut paraître importante par rapport à la taille occupée des unités de calcul dans cet exemple qui est de l'ordre de 2 %. Cependant, elle est compensée par les multiples possibilités de modification d'une application de manière dynamique : changement de contexte dynamique dans un SGM, rechargement d'un nouveau contexte, modification locale d'une unité de calcul. Ainsi, un simple changement de source d'image, dont le contexte de l'application est connu, permet de modifier dynamiquement le traitement des données sans surcoût en latence. Si ce n'est pas le cas, un temps d'attente de chargement est requis et dépend de la taille du contexte en mémoire.

7.4 Conclusion

Nous avons proposé dans ce chapitre une nouvelle architecture de réseau de communication en anneau appelée *Multi Data Flow Ring*, exploitant les différentes unités de routage et le système de mémoire de trames partagé. Son fonctionnement a été validé dans le cadre d'un application concrète pour la restitution d'image, typiquement utilisée dans des équipements de vision embarquée au CE-CTP Sagem.

Un premier prototype FPGA a été proposé afin d'évaluer les latences d'adaptations en chemin de données du réseau, les latences de chargement des contextes ainsi que la surface consommée. Des premiers résultats ont montré que les temps d'adaptation reste pertinents dans le cadre d'applications orientées flot de données pouvant être implémentées sous forme d'un pipeline d'unités de calcul. L'impact de ces latences d'adaptation est minimisé suivant la taille croissante du paquet de données. Une première évaluation en surface du réseau a montré que le surcoût de 10 % d'un composant FPGA de grande matrice (150k éléments logiques), est acceptable par rapport à la taille des unités de calcul utilisée. Ce surcoût est largement compensé par les possibilité d'adaptation dynamique d'une application en cours de fonctionnement. Cette adaptation peut être réalisée en basculant dynamiquement de contexte par modification de l'ID du paquet à traiter, en rechargeant complètement ou partiellement un contexte ou en modifiant localement les configurations d'une unité de calcul.

Chapitre 8

Conclusion générale et perspectives

8.1 Synthèse

La conception de l'architecture de calcul d'un système sur puce dédiée à la vision embarquée est un problème complexe. Elle nécessite de tenir compte du choix des capteurs d'image, de la disparité des types d'application, des contraintes de performance en temps et en surface tout en maintenant une consommation énergétique minimale.

Le développement d'architecture câblées n'est plus réaliste dans le contexte actuel où le marché de la vision embarquée portable est en constante évolution avec une variété de capteurs de résolutions d'image croissantes et une variété d'applications.

Dans un contexte industriel où les volumes de production sont faibles, un système de type *System-On-Programmable Chip* (SOPC) à base de technologie FPGA est une solution appropriée pour implémenter différentes applications de traitement d'image avec un faible coût. Ces applications évoluent en permanence pour améliorer la robustesse et augmenter les fonctionnalités d'un équipement de vision portable. Néanmoins, l'architecture implémentée dans le SOPC doit idéalement pouvoir s'adapter architecturalement avant synthèse dans le but de pouvoir adresser différentes familles de système de vision avec des contraintes différentes en terme de performance, d'encombrement et de consommation. Cette architecture nécessite également de s'adapter après synthèse afin d'implémenter efficacement une variété d'applications.

Différentes propositions d'architecture ont démontrées la puissance de calcul mais un verrou majeur persiste au niveau du système d'interconnexion qui n'est pas suffisamment adaptable architecturalement dans un contexte de système de vision multi-capteurs et multi-applications.

Dans cette thèse, nous proposons un *nouveau réseau de communication sur puce (NoC) pour un SoC dédié à la vision.* Ce réseau, adaptable avant synthèse, dispose de la capacité d'auto-adapter dynamiquement son chemin de donnée en fonction de différents flux de données transmis en parallèle. L'architecture de ce réseau est construite à partir deux types de routeurs : le routeur maître et le routeur esclave. Les noeuds maîtres sont connectés de façon indirecte par des noeuds esclaves. Dans ce réseau, les échanges principaux de paquets s'effectuent entre les noeuds maîtres et le traitement de ces paquets s'effectuent dans les noeuds esclaves contenant des unités de calcul.

Malgré l'hétérogénéité de capteurs, l'étude d'applications de visualisation d'image a montré la réutilisation de multiples opérations de traitement d'image entre chaque capteur. En effet, hormis la nécessité d'opération spécifiques propre à un capteur comme une opération de dématriçage, les différences majeures résident dans le séquencement de ces opérations. Ainsi, au niveau architecturale, le changement d'application repose essentiellement sur l'adaptation du chemin de communication entre des unités de calcul hétérogènes et sur la configuration de ces unités pour appliquer des coefficients de calcul différents par exemple.

En particulier, dans un contexte où des unités de calculs optimisées (IPs) ont été développées, la mise en oeuvre de plusieurs pipeline de ces unités est idéale pour traiter les images en flot de données avec le minimum de mémorisation et le minimum de latence.

Dans ce but, nous avons proposé d'adapter directement au niveau architectural, le chemin de donnée interne du routeur esclave afin de pouvoir garantir un ou plusieurs pipelines d'unités de calcul. Nous définissons ainsi différents mode de fonctionnement des routeurs avec différents chemin de données internes autorisant le traitement d'un flux de données de manière pipelinée ou parallèle.

Les images qui transitent dans le réseau sont paquetisées et nous proposons une nouvelle structure de l'en-tête du paquet afin de pouvoir identifier les sources de chaque image. Plus précisément, nous définissons la notion d'attributs permettant l'identification du

paquet de données sur le type de source d'image, le positionnement temporel de l'image associée et la dernière opération appliquée.

Une nouvelle méthode de communication des commandes sur les routeurs esclaves est proposée en associant directement des instructions avec la donnée à traiter. Ces instructions correspondent à un séquencement d'opération à appliquer sur la donnée en fonction de l'application à implémenter.

A partir de notre nouvelle structure de paquet, nous proposons une méthode d'aiguillage, appelée *Split-Wormhole switching*, adaptée à notre routeur esclave. L'algorithme de routage dynamique associée à ce routeur permet de décider de la modification du chemin en fonction de l'adresse de destination et des instructions contenues dans l'en-tête.

Pour réaliser le routeur esclave appelé *Data Flow Router* (DFR), nous avons défini, implémentée et évaluée une nouvelle architecture capable d'adapter dynamiquement le chemin de donnée interne sur plusieurs voies de communication en parallèle. Cette adaptation s'effectue de façcon à réaliser de la manière la plus efficace possible les différents modes d'exécution requis entre les opérations dans une application donnée, dans le but de s'approcher des performances d'une solution dédiée point-à-point, tout en ayant une flexibilité dans le chemin de données. L'implémentation de cette proposition démontre les performances en temps d'adaptation, la simplicité des communication des instructions et la consommation en surface raisonnable dans le contexte d'une architecture utilisant des unités de calcul à grain-moyen.

Ces instructions sont spécifiées par un contexte mémoire contenu dans les routeurs maîtres, appelés *Stream Gate Managers* (SGM).

Afin d'implémenter efficacement des opérations de traitement d'image temporelles, nous proposons une nouvelle méthode de stockage de trames en mémoire. Cette méthode est basée sur un partitionnement de la mémoire en plusieurs emplacements physiques et un adressage de ces emplacements de manière indirecte par l'intermédiaire d'indicateurs caractérisant une trame dans une chaine de traitement. Basée sur cette méthode, nous proposons l'architecture de système mémoire dédiée au stockage des trames, appelée *Frame Buffer System* (FBS) et composée d'un banc mémoire associé à une surcouche d'abstraction adaptative.

Nous proposons une organisation mémoire adaptée à notre réseau permettant d'implémenter efficament diverses applications. Dans cette organisation, chaque routeur maître, contenant des mémoires capables de stocker des lignes d'image, dispose d'un accès sur une mémoire partagée dédiée aux trames d'image. L'évaluation des architectures proposées pour le SGM et le FBS sur une cible FPGA, démontre les coûts en surface raisonnables permettant d'atteindre une plus grande souplesse dans la gestion des trames stockées. Les mesures en temps ont permis de confirmer que l'adaptation reste performante dans le cadre de la proposition du réseau.

Afin de valider notre proposition, nous proposons une nouvelle architecture de réseau de communication en anneau appelée *Multi Data Flow Ring*, exploitant les différentes unités de routage et le système de mémoire de trames partagée. Son fonctionnement a été validée dans le cadre d'une application concrète pour la visualisation d'image, typiquement utilisée dans des équipements de vision embarquée au CE-CTP Sagem. Un prototype FPGA a été proposé afin d'évaluer les latences d'adaptations en chemin de données du réseau, les latences de chargement des contextes ainsi que la surface consommée. Des premiers résultats ont montré que les temps d'adaptation reste pertinents dans le cadre d'applications orientées flot de données pouvant être implémentées sous forme d'un pipeline d'unités de calcul. L'impact de ces latences d'adaptation est minimisé suivant la taille croissante du paquet de données. Une première évaluation en surface du réseau a montré que le surcoût de 10 % d'un composant FPGA de grande matrice (150k éléments logiques), est acceptable par rapport à la taille des unités de calcul utilisée. Ce surcoût est largement compensé par les possibilités d'adaptation dynamique d'une application en cours de fonctionnement. Cette adaptation peut être réalisée en basculant dynamiquement de contexte par modification de l'ID du paquet à traiter, en rechargeant complètement ou partiellement un contexte ou en modifiant localement les configurations d'une unité de calcul.

8.2 Perspectives

Les perspectives de ces travaux sont conséquentes en terme d'application et de conception d'une électronique innovante de système de vision embarquée portable multi-capteurs.

Dans un premier temps, un travail d'exploration architecturale autour de l'architecture MDFR proposée est important pour évaluer les performances temporelles dans le cadre de diverses applications en vision embarquée. La topologie du réseau peut être également modifiée sous réserve des contraintes de construction spécifiées dans ce manuscrit.

Les évaluations architecturales proposées dans cette thèse n'ont été réalisées que pour de faibles résolutions d'image en raison de la limitation en espace mémoire sur puce FPGA. Une démonstration complète avec une pleine résolution serait envisageable en adaptant l'architecture MDFR avec le FBS réalisé avec une mémoire RAM externe. D'un point de vue contrôle global des chargements de contexte, une unité programmable externe complèterait le système proposé en se chargeant de la génération des en-têtes qui encapsulent les flux de pixels entrants dans le MDFR, et en gérant de manière centralisée l'injection des flux de pixels externe au MDFR.

D'autre part, l'architecture MDFR proposée n'a été conçue que dans un cadre d'utilisation d'unité de calcul dédiées, orientées flot de données, et pipelinables. Il serait pertinent d'étudier des extensions de cette architecture dans le cadre d'utilisation d'unités de calcul de types processeurs élementaires programmables. Se trouvent alors posé la réflexion sur les méthodes de chargements de code instructions spécifiques à chaque unité, des techniques de synchronisation des opérations, etc.

Par ailleurs, la mise au point d'un environnement logiciel de planification est nécessaire afin de construire un réseau de communication pertinent et générer automatiquement les instructions à charger en fonction des applications à implémenter. Le couplage de l'outil SynDex de l'INRIA, pour planifier le mapping des chemins de données, et l'outil SynDex-IC de l'ESIEE, pour générer des unités de calcul, permettrait d'atteindre ce but par exemple.

Enfin, d'un point de vue intégration dans un système de vision embarquée portable, ces travaux ouvrent des opportunités dans la conception de système à capteurs interchangeables et de système intégrant le capteur comme une unité de calcul dans le réseau. En effet, il est tout à fait envisageable de définir une électronique de proximité dans le capteur, capable de générer des contextes spécifiques, propres au capteur, à charger dans l'architecture de réseau qui est capable de s'adapter dynamiquement.

Bibliographie

[1] Junichi Nakamura. *Image sensors and signal processing for digital still cameras.* CRC Press, 2006.

[2] William L. Wolfe. *Introduction to infrared system design.* SPIE Press, 1996.

[3] Joseph Caniou. *Passive infrared detection : theory and applications.* Springer, 1999.

[4] Honghao Ji and P.A. Abshire. A cmos image sensor for low light applications. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, page 4 pp., 0-0 2006.

[5] M. Kumar, E.O. Morales, J.E. Adams, and Wei Hao. New digital camera sensor architecture for low light imaging. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 2681 –2684, 2009.

[6] Jeroen Wehmeijer and Bert van Geest. High-speed imaging : Image intensification. *Nature Photonics*, 4 :152–153, 2010.

[7] P. van der Wolf and T. Henriksson. Video processing requirements on soc infrastructures. pages 1124 –1125, mar. 2008.

[8] Alan Conrad Bovik. *Handbook of image and video processing.* Academic Press, 2005.

[9] Scott Hauck and Andre DeHon. *Reconfigurable Computing : The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann, 2007.

[10] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing.* Springer, 2007.

[11] Junming Xu. *Topological Structure and Analysis of Interconnection Networks.* Kluwer Academic Publishers, 2001.

[12] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. *Design, Automation and Test in Europe Conference and Exhibition*, 0 :250, 2000.

[13] W.J. Dally and B. Towles. Route packets, not wires : on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684 – 689, 2001.

[14] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Elsevier, 2004.

[15] Nikolay Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* PhD thesis, University of Twente, 2007. CTIT Ph.D.-thesis series No. 06-91 ISBN 90-365-2410-5.

[16] Hamid Aghajan and Andrea Cavallaro. *Multi-Camera Networks : Principles and Applications.* Academic Press - Elsevier, 2009.

[17] Jan van der Horst, Rien van Leeuwen, Harry Broers, Richard Kleihorst, and Pieter Jonker. A real-time stereo smartcam, using fpga, simd and vliw. In *Proc. 2nd Workshop on Applications of Computer Vision (Graz, May 12), Austria*, pages 1–8, 2006.

[18] Teng-Yuan Cheng, Tsung-Huang Chen, J.C. Chen, and Shao-Yi Chien. Coarse-grained reconfigurable image stream processor architecture for high-definition cameras and camcorders. In *SoC Design Conference (ISOCC), 2010 International*, pages 95 –98, nov. 2010.

[19] Nicolas Ngan, Geoffroy Marpeaux, Eva Dokladalova, Mohamed Akil, and Francois Contou-Carrére. Memory system for a dynamically adaptable pixel stream architecture. In *IEEE International Conference on Field Programmable Logic and Applications (FPL'10)*, 2010.

[20] Nicolas Ngan, Eva Dokladalova, Mohamed Akil, and François Contou-carrere. Dynamically adaptable architecture for real-time video processing. In *IEEE International Symposium on Circuits and Systems (ISCAS'10)*, 2010.

[21] Dave Litwiller. Ccd vs cmos : Facts and fictions. *Photonics Spectra*, 2001.

[22] E.R. Fossum. Cmos image sensors : electronic camera-on-a-chip. *Electron Devices, IEEE Transactions on*, 44(10) :1689 –1698, oct 1997.

[23] B.S. Carlson. Comparison of modern ccd and cmos image sensor technologies and systems for low resolution imaging. In *Sensors, 2002. Proceedings of IEEE*, volume 1, pages 171 – 176 vol.1, 2002.

[24] Bryce E. Bayer. Color imaging array, 1976.

[25] David Alleyson. 30 years of demosaicing. *Traitement du Signal*, 21 :561–581, 2004.

[26] A. Rabner and Y. Shacham-Diamand. Electron-bombarded cmos image sensor in single photon imaging mode. *Sensors Journal, IEEE*, 11(1) :186 –193, jan. 2011.

[27] R. Lenggenhager, H. Baltes, J. Peer, and M. Forster. Thermoelectric infrared sensors by cmos technology. *Electron Device Letters, IEEE*, 13(9) :454 –456, sep 1992.

[28] C. Marshall, T. Parker, and T. White. Infrared sensor technology. In *Engineering in Medicine and Biology Society, 1995., IEEE 17th Annual Conference*, volume 2, pages 1715 –1716 vol.2, sep 1995.

[29] Rastislav Lukac. Single-sensor imaging in consumer digital cameras : a survey of recent advances and future directions. *Journal of Real-Time Image Processing*, 1 :45–52, 2006.

[30] Rick S. Blum and Zheng Liu. *Multi-Sensor Image Fusion and Its Applications*. CRC, 2005.

[31] Tania Stathaki. *Image Fusion : Algorithms and Applications*. Academic Press, 2008.

[32] Christophe Hennequin. *Etude et réalisation d'un calculateur temps réel embarqué pour la détection de petits objets dans des séquences d'images multi-échelles*. PhD thesis, Université de Bourgogne, 2008.

[33] Fan Yang and Michel Paindavoine. Implementation of a rbf neural network on embedded systems : Real time face tracking and identity verification. *IEEE Transactions on Neural Networks*, Vol.14 (N°5) :pp. 1162–1175, 2003.

[34] Nicolas Farrugia. *Architectures parallèles pour l'analyse de visages embarquée*. PhD thesis, Université de Bourgogne, Laboratoire Le2i, 2008.

[35] F. Heitz and P. Bouthemy. Motion estimation and segmentation using a global bayesian approach. In *Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on*, pages 2305 –2308 vol.4, apr 1990.

[36] Simon Khaskelman. Présentation d'un chaîne de traitements vidéos (cours ensta 2008). Technical report, Sagem, CE-CTP, EUROSAE, 2008.

[37] Hui xin Zhou, Rui Lai, Shang qian Liu, and Guang Jiang. New improved non-uniformity correction for infrared focal plane arrays. *Optics Communications*, 245(1-6) :49 – 53, 2005.

[38] Harold Phelippeau. *Méthodes et algorithmes de dématriçage et de filtrage du bruit pour la photographie numérique*. PhD thesis, Université Paris-Est, 2009.

[39] A. Buades, B. Coll, and J. M. Morel. A review of image denoising algorithms, with a new one. *Simul*, 4 :490–530, 2005.

[40] Fredo Durand and Julie Dorsey. Fast bilatral filtering for the display of high-dynamic-range images. *ACM*, 2002.

[41] F. Jin, P. Fieguth, L. Winger, and E. Jernigan. Adaptive wiener filtering of noisy images and image sequences. In *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, volume 3, pages III – 349–52 vol.2, sept. 2003.

[42] Jianqing Fan and Ja-Yong Koo. Wavelet deconvolution. *Information Theory, IEEE Transactions on*, 48(3) :734 –747, mar 2002.

[43] William Hadley Richardson. Bayesian-based iterative method of image restoration. *JOSA*, 62 :55–59, 1972.

[44] Wen Wang, Bo Li, Jin Zheng, Shu Xian, and Jing Wang. A fast multi-scale retinex algorithm for color image enhancement. In *Wavelet Analysis and Pattern Recognition, 2008. ICWAPR '08. International Conference on*, volume 1, pages 80 –85, aug. 2008.

[45] K.H. Abas, O. Ono, and Z. Ibrahim. Enhancement of infrared-based image identification system for security robots by image decomposition. In *Autonomous Robots*

and Agents, 2009. ICARA 2009. 4th International Conference on, pages 398 –402, feb. 2009.

[46] Heechang Kim, Sangjun Park, Jin Wang, Yonghoon Kim, and Jechang Jeong. Advanced bilinear image interpolation based on edge features. In *Advances in Multimedia, 2009. MMEDIA '09. First International Conference on*, pages 33 – 36, july 2009.

[47] J. Prades-Nebot, A. Albiol, and C. Bachiller. Enhanced b-spline interpolation of images. In *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, pages 289 –293 vol.3, oct 1998.

[48] M. Unser, A. Aldroubi, and M. Eden. Fast b-spline transforms for continuous image representation and interpolation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 13(3) :277 –285, mar 1991.

[49] Sung Cheol Park, Min Kyu Park, and Moon Gi Kang. Super-resolution image reconstruction : a technical overview. *Signal Processing Magazine, IEEE*, 20(3) :21 – 36, may 2003.

[50] S. Chikamatsu, T. Nakaya, M. Kouda, N. Kuroki, T. Hirose, and M. Numa. Super-resolution technique for thermography with dual-camera system. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1895 –1898, 30 2010-june 2 2010.

[51] V. Petrovic and T. Cootes. Objectively optimised multisensor image fusion. In *Information Fusion, 2006 9th International Conference on*, pages 1 –7, 2006.

[52] Arnold Meijster. *Efficient Sequential and Parallel Algorithms for Morphological Image Processing*. PhD thesis, University Groningen, 2004.

[53] Nicolas Ngan, F. Contou-Carrère, B. Marcon, S. Guerin, Eva Dokládalova, and Mohamed Akil. Efficient hardware implementation of connected component tree algorithm . In *Workshop on Design and Architectures For Signal and Image Processing*, 2007.

[54] Nicolas Ngan, Eva Dokladalova, Mohamed Akil, and Francois Contou-Carrere. Fast and efficient fpga implementation of connected operators. *Journal of Systems Architecture*, 2009.

[55] International Technology Roadmap for Semiconductors. Irts reports. Technical report, http ://www.itrs.net, 2011.

[56] Shahram Zahirazami. *Architecture Reconfigurable : Conception et Evaluation d'un système reconfigurable pour le traitement bas niveau d'images en temps réel*. PhD thesis, Université de Paris Sud, 1999.

[57] Hongtu Jiang. *Design Issues in VLSI Implementation of Image Processing Hardware Accelerator*. PhD thesis, Department of Electroscience, Lund University, 2007.

[58] Viorela Simona. *VLSI Architecture For Motion Estimation In Underwater Imaging*. PhD thesis, Universitat de Girona, 2005.

[59] Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann, 2006.

[60] IBM Corporation. *Book E : Enhanced PowerPC Architecture, Third Edition*. 2002.

[61] Intel. Intel atom processor specifications (intel.com). Technical report, 2010.

[62] ARM. The arm cortex-a9 processors (www.arm.com). Technical report, 2010.

[63] ARM. Cortex-a9 mpcore technical reference manual (www.arm.com). Technical report, 2010.

[64] Lionel Torres, Pascal Benoit, G. SASSATELLI, M. ROBERT, D. PUSCINI, and F. CLERMIDY. *"An Introduction to Multiprocessor System-on-Chip : Trends and Challenges" Multiprocessor System-on-Chip Hardware Design and Tool Integration Hübner, Michael; Becker, Jürgen ISBN : 978-1-4419-6459-5 pp. 1-24*. SPRINGER, 2011.

[65] Texas Instruments. Tms320c6670 multicore fixed and floating-point system-on-chip (rev. a) (www.ti.com). Technical report, 2011.

[66] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 140–150, New York, NY, USA, 1983. ACM.

[67] Vincent Brost, Fan Yang, Michel Paindavoine, and Nicolas Farrugia. Multiple modular vliw processors based on fpga. *Journal of Electronic Imaging*, Article ID 24163, 2007.

[68] Peter Kollig, Colin Osborne, and Tomas Henriksson. Heterogeneous multi-core platform for consumer multimedia applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1254–1259, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[69] Texas Instruments. Tms320dm365 digital media system-on-chip (rev. e) (www.ti.com). Technical report, 2011.

[70] ARM. Arm processor instruction set architecture (www.arm.com). Technical report, 2009.

[71] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16 :42–50, August 1996.

[72] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20 :85–95, March 2000.

[73] Sourav Chatterji, Manikandan Narayanan, Jason Duell, and Leonid Oliker. Performance evaluation of two emerging media processors : Viram and imagine. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 229.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[74] Carlos Alba Pinto, Aleksandar Beric, Satendra Pal Singh, and Sachin Farfade. Hiveflex-video vsp1 : Video signal processing architecture for video coding and post-processing. *Multimedia, International Symposium on*, 0 :493–500, 2006.

[75] S. Kyo, S. Nomoto, and S. Okazaki. Mapping schemes of image recognition tasks onto highly parallel simd/mimd processors. In *Distributed Smart Cameras, 2009. ICDSC 2009. Third ACM/IEEE International Conference on*, pages 1 –6, 30 2009-sept. 2 2009.

[76] A. Abbo, R. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, and M. Heijligers. Xetal-ii : A 107 gops, 600mw massively-parallel processor for video scene analysis. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 270 –602, feb. 2007.

[77] Yu Pu, Yifan He, Zhenyu Ye, S.M. Londono, A.A. Abbo, R. Kleihorst, and H. Corporaal. From xetal-ii to xetal-pro : On the road toward an ultralow-energy and high-throughput simd processor. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(4) :472 –484, april 2011.

[78] Jason Sanders and Edward Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[79] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla : A unified graphics and computing architecture. *Micro, IEEE*, 28(2) :39 –55, march-april 2008.

[80] ARM. Mali-400 mp (www.arm.com). Technical report, 2010.

[81] Imagination Technologies. Powervr mbx technology overview (www.imgtec.com). Technical report, 2009.

[82] Texas Instruments. Omap3530/25 applications processor (rev. f). Technical report, 2009.

[83] B.K. Khailany, T. Williams, J. Lin, E.P. Long, M. Rygh, D.W. Tovey, and W.J. Dally. A programmable 512 gops stream processor for signal, image, and video processing. *Solid-State Circuits, IEEE Journal of*, 43(1) :202 –213, jan. 2008.

[84] Nvidia. The benefits of multiple cpu cores in mobile devices (www.nvidia.com). Technical report, 2010.

[85] Reiner Hartenstein. A decade of reconfigurable computing : A visionary retrospective. 2001.

[86] Armando Astarloa. Tornado : A self-reconfiguration control system for core-based multiprocessor csopcs. *ELSEVIER*, 2007.

[87] F.J. Gomez-Arribas I. Gonzalez, S. Lopez-Buedo. Implementation of secure applications in self-reconfigurable systems. *ELSEVIER*, 2007.

[88] Pascal Benoit, Lionel Torres, G. SASSATELLI, and N. SAINT-JEAN. Run-time mapping for dynamic reconfiguration management in embedded systems. *International Journal of Embedded Systems, Interdescience*, 2010.

[89] Katherine Compton. Programming architectures for run-time reconfigurable systems. Master's thesis, Department of ECE Northwestern University Evanston, IL USA, 1999.

[90] Lattice. Latticexp2 family data sheet. Technical report, 2011.

[91] C. Claus and J. Zeppenfeld. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *DATE 2007*, 2007.

[92] Intel. Premières puces configurables à base de processeur intel atom (www.intel.com). Technical report, 2010.

[93] Xilinx. Zynq-7000 extensible processing platform (www.xilinx.com). Technical report, 2011.

[94] J.C. Chen and Shao-Yi Chien. Crisp : Coarse-grained reconfigurable image stream processor for digital still cameras and camcorders. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(9) :1223 –1236, sept. 2008.

[95] Tsung-Huang Chen, J.C. Chen, Teng-Yuan Cheng, and Shao-Yi Chien. Crisp-ds : Dual-stream coarse-grained reconfigurable image stream processor for hd digital camcorders and digital still cameras. In *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, pages 193 –196, nov. 2009.

[96] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 66–81, Berlin, Heidelberg, 2008. Springer-Verlag.

[97] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *Proceedings of the 3rd international conference on Reconfigurable computing : architectures, tools and applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[98] Didier Demigny. *Méthodes et architectures pour le TSI en temps-réel*. Lavoisier, 2001.

[99] Ryad Bourguiba, Didier Demigny, and Lounis Kessal. Architecture reconfigurable dynamiquement, application au traitement d'image. 1999.

[100] Nicholas Peter Sedcole. *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing.* PhD thesis, Department of Electrical and Electronic Engineering Imperial College of Science, Technology and Medicine, University of London, 2006.

[101] Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gérard Gaillat, Olivier Ruch, and Pascal Gauget. Definition and simd implementation of a multi-processing architecture approach on fpga. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 610–615, New York, NY, USA, 2008. ACM.

[102] ALTERA. Stratix v device handbook, 2011.

[103] Achronix. Speedster22i-hp (www.achronix.com). Technical report, 2011.

[104] David J. Katz and Rick Gentile. *Embedded Media Processing.* Elsevier, 2006.

[105] Wayne Wolf. *High-Performance Embedded Computing.* Elsevier, 2007.

[106] Arteris. From "bus" and "crossbar" to "network-on-chip" (www.arteris.com). Technical report, 2009.

[107] ARM. Amba open specifications (www.arm.com). Technical report, 2011.

[108] Emmanuel Casseau. Roma : Reconfigurable operators for multimedia applications, projet anr « architectures du futur 2006 ». In *Colloque « Systèmes embarqués, sécurité et sûreté de fonctionnement », Toulouse*, 2010.

[109] Terry Tao Ye. *On-Chip Multiprocessor Communication Network Design And Analysis.* PhD thesis, Standford University, 2003.

[110] Julien Denoulet. *Architectures massivement parallèles de systèmes sur circuits (SoC) pour le traitement de flux vidéos.* PhD thesis, Université Paris XI Orsay, 2004.

[111] E. Salminen, A. Kulmala, and T.D. Hamalainen. On network-on-chip comparison. pages 503 –510, aug. 2007.

[112] J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot. A fpga partial reconfiguration design approach for cognitive radio based on noc architecture. In *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, pages 355 –358, 2008.

[113] C. Hilton and B. Nelson. Pnoc : a flexible circuit-switched noc for fpga-based systems. *Computers and Digital Techniques, IEE Proceedings* -, 153(3) :181 – 188, may. 2006.

[114] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes : an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1) :69 – 93, 2004.

[115] V. Fresse, J. Tan, and F. Rousseau. Exploration of an adaptive noc architecture on fpga dedicated to multi and hysperspectral algorithm for art authentication. In *IPTA 2010*, pages 529 –534, jul. 2010.

[116] S. Le Beux, G. Nicolescu, G. Bois, Y. Bouchebaba, M. Langevin, and P. Paulin. Optimizing configuration and application mapping for mpsoc architectures. pages 474 –481, jul. 2009.

[117] Linlin Zhang, V. Fresse, M. Khalid, D. Houzet, M. Ahmadi, A.-C. Legrand, and V. Fischer. Evaluation of noc dedicated to multispectral image data communication. pages 1 –4, jul. 2009.

[118] Donghyun Kim, Kwanho Kim, Joo-Young Kim, Seungjin Lee, and Hoi-Jun Yoo. An 81.6 gops object recognition processor based on noc and visual image processing memory. pages 443 –446, sep. 2007.

[119] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda. Cunoc : A scalable dynamic noc for dynamically reconfigurable fpgas. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 753 –756, 2007.

[120] Slavisa Jovanovic. *Architecture reconfigurable de systèmes embarqué auto-organisé*. PhD thesis, Université Henry Poincaré - Nancy 1, Laboratoire d'Instrumentation Electronique de Nancy (LIEN), 2009.

[121] M. Majer, C. Bobda, A. Ahmadinia, and J. Teich. Packet routing in dynamically changing networks on chip. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 154b – 154b, 2005.

[122] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen. Dynoc : A dynamic infrastructure for communication in dynamically reconfugurable devices. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 153 – 158, 2005.

[123] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *Computers and Digital Techniques, IEE Proceedings* -, 152(4) :467 – 472, jul. 2005.

[124] D. Gohringer, M. Hubner, T. Perschke, and J. Becker. New dimensions for multiprocessor architectures : On demand heterogeneity, infrastructure and performance through reconfigurability ; the rampsoc approach. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 495 –498, 2008.

[125] D. Gohringer, Bin Liu, M. Hubner, and J. Becker. Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit - and a packet-switching communication protocol. pages 320 –325, aug. 2009.

[126] Daniel Lüdtke, Dietmar Tutsch, and Günter Hommel. An analyzable on-chip network architecture for embedded systems. In Günter Hommel and Sheng Huanye, editors, *Embedded Systems – Modeling, Technology, and Applications*, pages 63–72. Springer Netherlands, 2006.

[127] Christophe Clienti, Serge Beucher, and Michel Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. 2008.

[128] F.A. Samman, T. Hollstein, and M. Glesner. Multicast parallel pipeline router architecture for network-on-chip. pages 1396 –1401, mar. 2008.

[129] F.A. Samman, T. Hollstein, and M. Glesner. Flexible parallel pipeline network-on-chip based on dynamic packet identity management. pages 1 –8, apr. 2008.

[130] Stephan Bourduas. *Modeling, Evaluation, and Implementation of Ring-Based Interconnects for Network-on-Chip*. PhD thesis, McGill University, Faculty of Engineering, Department of Electrical and Computer Engineering, 2008.

[131] Manuel Saldana, Lesley Shannon, and Paul Chow. The routability of multiprocessor network topologies in fpgas. 2006.

[132] Shuwei Bai, Qingguo Zhou, Rui Zhou, and Lian Li. Barrier synchronization for cell multi-processor architecture. In *Proc. First IEEE International Conference on Ubi-Media Computing*, pages 155–158, July 2008.

[133] Creative. X-fi ring architecture (www.creative.com).

[134] Lionel Torres. The systolic ring : A dynamically reconfigurable architecture for socs and embedded systems. In *Systems on Chips workshop, Tampere University of Technology, Finlande*, 2001.

[135] Pascal Benoit. *Architectures des accélérateurs de traitements flexibles pour les systèmes sur puce*. PhD thesis, Université Montpellier II (LIRMM), 2004.

[136] F. Clermidy, R. Lemaire, Y. Thonnart, and P. Vivet. A communication and configuration controller for noc based reconfigurable data flow architecture. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 153 –162, May 2009.

[137] Lars Braun, Diana Gohringer, Thomas Perschke, Volker Schatz, Michael Hubner, and Jargen Becker. Adaptive real-time image processing exploiting two dimensional reconfigurable architecture. *Journal of Real-Time Image Processing*, 4 :109–125, 2009. 10.1007/s11554-008-0095-8.

[138] D. Wiklund and Dake Liu. Socbus : switched network on chip for hard real time embedded systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., april 2003.

[139] Ge-Ming Chiu. The odd-even turn model for adaptive routing. *Parallel and Distributed Systems, IEEE Transactions on*, 11(7) :729 –738, jul 2000.

[140] T. Schonwald, J. Zimmermann, O. Bringmann, and W. Rosenstiel. Fully adaptive fault-tolerant routing algorithm for network-on-chip architectures. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 527 –534, aug. 2007.

[141] S.A. Asghari, H. Pedram, and M. Khademi. A flexible design of network on chip router based on handshaking communication mechanism. pages 225 –230, oct. 2009.

[142] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. Xpipes : a latency insensitive parameterized network-on-chip architecture for multiprocessor socs. pages 536 – 539, oct. 2003.

[143] Henrique C. Freitas and Philippe O.A. Naveaux. Noc architecture design for multi-cluster chips. 2008.

[144] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on fpgas. In *FPL '02 : Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 795–805, London, UK, 2002. Springer-Verlag.

[145] Seung Eun Lee and Nader Bagherzadeh. Increasing the throughput of an adaptive router in network-on-chip (noc). In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 82–87, New York, NY, USA, 2006. ACM.

[146] J. Hilgenstock, K. Hermann, and P. Pirsch. Memory organization of a single-chip video signal processing system with embedded dram. pages 42 –45, mar 1999.

[147] Anders Kjaer-Nielsen, Lars Baunegaard With Jensen, Anders Stengaard Sorensen, and Norbert Kruger. A real-time embedded system for stereo vision preprocessing using an fpga. *Reconfigurable Computing and FPGAs, International Conference on*, 0 :37–42, 2008.

[148] Video Electronics Standards Association (VESA). http ://www.vesa.org/.

[149] David Tawei Wang. *Modern DRAM Memory Systems : Performance Analysis And Scheduling Algorithm.* PhD thesis, University of Maryland, College Park, 2005.

[150] Jeyran Hezavei, N. Vijaykrishnan, and M. J. Irwin. A comparative study of power efficient sram designs. In *Proceedings of the 10th Great Lakes symposium on VLSI*, GLSVLSI '00, pages 117–122, New York, NY, USA, 2000. ACM.

[151] ALTERA. Wp-01134-1.0 : Boosting system performance with external memory solutions. Technical report, 2010.

[152] ALTERA. External memory interface handbook. Technical report, 2011.

[153] SAMSUNG. Qdr ii sram : High-bandwidth memory for advanced network equipment, 2009.

[154] MICRON. http ://www.micron.com/.

[155] Eero Aho. *Design and Implementation of Parallel Memory Architecture*. PhD thesis, Tampere University of Technology, 2006.

[156] XILINX. Xilinx ug388 spartan-6 fpga memory controller user guide, 2011.